



O₂Kit

User Manual

Release 5.0 - April 1998



Information in this document is subject to change without notice and should not be construed as a commitment by O₂ Technology.

The software described in this document is delivered under a license or nondisclosure agreement.

The software can only be used or copied in accordance with the terms of the agreement. It is against the law to copy this software on magnetic tape, disk, or any other medium for any purpose other than the purchaser's own use.

Copyright © 1998 O₂ Technology.

All rights reserved. No part of this publication can be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopy without prior written permission of O₂ Technology.

O₂ and O₂API, O₂C, O₂DBAccess, O₂Engine, O₂Graph, O₂Kit, O₂Look, O₂Store, O₂Tools are registered trademarks of O₂ Technology.

SQL and AIX are registered trademarks of International Business Machines Corporation.

Sun, SunOS and SOLARIS are registered trademarks of Sun Microsystems, Inc.

X Window System is a registered trademark of the Massachusetts Institute of Technology.

Unix is a registered trademark of Unix System Laboratories, Inc.

HPUX is a registered trademark of Hewlett-Packard Company.

BOSX is a registered trademark of Bull S.A.

IRIX is a registered trademark of Siemens Nixdorf, A.G.

NeXTStep is a registered trademark of the NeXT Computer, Inc.

Purify, Quantify are registered trademarks of Pure Software Inc.

Windows is a registered trademark of Microsoft Corporation.

All other company or product names quoted are trademarks or registered trademarks of their respective trademark holders.

Who should read this manual

O₂Kit provides the programmer with predefined classes and methods for faster development of user applications. This manual details the Date, Text, Bitmap, and Image classes, and describes the Hyper Image facility with which icons may be superimposed on any graphical display. How to customize application dialog boxes used in conjunction with O₂Look is also described.

Other documents available are outlined, click below.

See [O2 Documentation set](#).



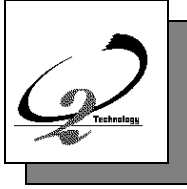


TABLE OF CONTENTS

This manual is divided into the following sections:

- 1.1 - Introduction
- 1.2 - The O₂Kit Programmer's Toolbox
- 1.3 - The Hyper Facility
- 1.4 - The Widget Editor



TABLE OF CONTENTS

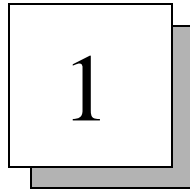
1	Introduction	9
	1.1 System overview	10
	1.2 Manual overview.....	12
2	O2Kit Programmer's Toolbox	15
	2.1 The class Date	16
	Methods of class Date	16
	Using the specific editor date.....	22
	Class Date: schema	25
	2.2 Dialogue boxes.....	26
	The class Box	26
	Methods of class Box	27
	The class Component	33
	The class Dialog_box.....	41
	Two examples	42
	Graphic resources for dialogue boxes.....	48
	Dialogue boxes: schema.....	54
	2.3 The class Text.....	58
	Methods of the class Text	59
	An example	60
	Class Text: schema.....	62
	2.4 The class Bitmap.....	62
	Methods of class Bitmap.....	63
	Class Bitmap: schema	64
	2.5 The class Image.....	64
	Methods of the class Image.....	64
	The specific editor image	65
	image resources	66
3	The Hyper Facility	69
	3.1 Introduction	70

TABLE OF CONTENTS

	3.2 The background	72
	3.3 Data structures	73
	3.4 Presentation and display	75
	3.5 Programming example	78
	3.6 Interactions	83
4	The Widget Editor	87
	4.1 Introduction	88
	4.2 The class WidgetDialoger	88
	4.3 Methods of the class WidgetDialoger	89
	4.4 An example of the user interface	91
	INDEX	95



TABLE OF CONTENTS



Introduction

GENERAL OVERVIEW OF THE O₂KIT SERVICE

Congratulations! You are now a user of the O₂Kit service!

This chapter is divided into the following sections:

- [System overview](#)
- [Manual overview](#)

1.1 System overview

The system architecture of O₂ is illustrated in Figure 1.1.

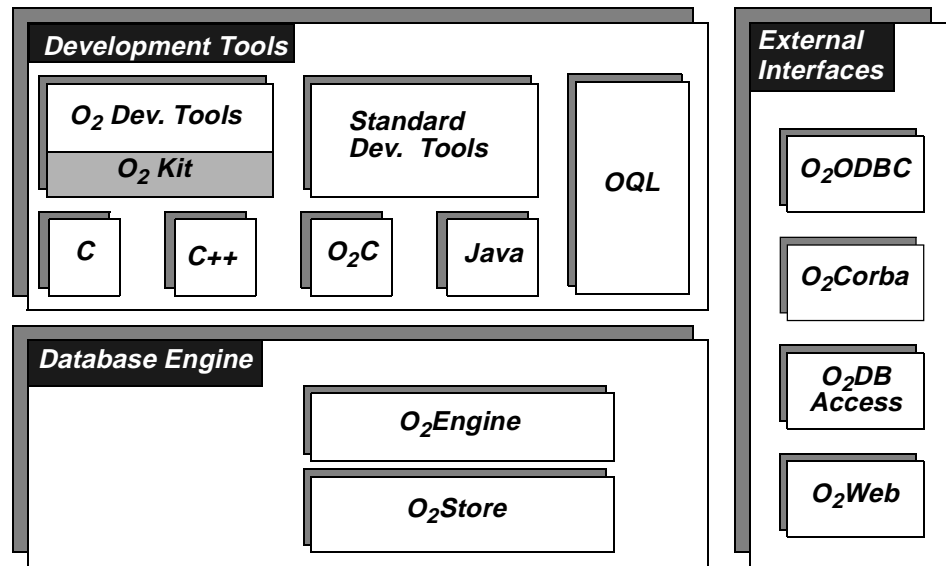


Figure 1.1: O₂ System Architecture

The O₂ system can be viewed as consisting of three components. The Database Engine provides all the features of a Database system and an object-oriented system. This engine is accessed with Development Tools, such as various programming languages, O₂ development tools and any standard development tool. Numerous External Interfaces are provided. All encompassing, O₂ is a versatile, portable, distributed, high-performance dynamic object-oriented database system.

Database Engine:

- O₂Store The database management system provides low level facilities, through O₂Store API, to access and manage a database: disk volumes, files, records, indices and transactions.
- O₂Engine The object database engine provides direct control of schemas, classes, objects and transactions, through O₂Engine API. It provides full text indexing and search capabilities with O₂Search and spatial indexing and retrieval capabilities with O₂Spatial. It includes a Notification manager for informing other clients connected to the same O₂ server that an event has occurred, a Version manager for handling multiple object versions and a Replication API for synchronizing multiple copies of an O₂ system.

System overview

Programming Languages:

O₂ objects may be created and managed using the following programming languages, utilizing all the features available with O₂ (persistence, collection management, transaction management, OQL queries, etc.)

- C O₂ functions can be invoked by C programs.
- C++ ODMG compliant C++ binding.
- Java ODMG compliant Java binding.
- O₂C A powerful and elegant object-oriented fourth generation language specialized for easy development of object database applications.
- OQL ODMG standard, easy-to-use SQL-like object query language with special features for dealing with complex O₂ objects and methods.

O₂ Development Tools:

- O₂Graph Create, modify and edit any type of object graph.
- O₂Look Design and develop graphical user interfaces, provides interactive manipulation of complex and multimedia objects.
- O₂Kit Library of predefined classes and methods for faster development of user applications.
- O₂Tools Complete graphical programming environment to design and develop O₂ database applications.

Standard Development Tools:

All standard programming languages can be used with standard environments (e.g. Visual C++, Sun Sparcworks).

External Interfaces:

- O₂Corba Create an O₂/Orbix server to access an O₂ database with CORBA.
- O₂DBAccess Connect O₂ applications to relational databases on remote hosts and invoke SQL statements.
- O₂ODBC Connect remote ODBC client applications to O₂ databases.
- O₂Web Create an O₂ World Wide Web server to access an O₂ database through the internet network.

1.2 Manual overview

This manual is divided into the following chapters:

- ***Chapter 1 - The O₂ System Overview***

- ***Chapter 2 - The O₂Kit Programmer's Toolbox***

This chapter details the various classes and methods of the O₂ Kit service which you may use, for instance `Data`, `text`, `Bitmap`, `Image`.

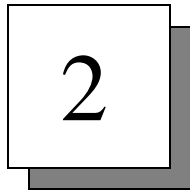
- ***Chapter 3 - The Hyper Facility***

This chapter describes how to use the hyper editor of O₂ Look to display icons which may serve as active buttons.

- ***Chapter 4 - The Widget Editor***

This chapter describes the O₂ Look Widget Editor which enables you to insert any Motif window into an O₂ Look presentation.

Manual overview



O₂Kit Programmer's Toolbox

For some types of often-used objects, the programmer will want to use predefined classes, with specific methods, instead of reinventing the wheel; the most common examples of this are `date` and `text`.

O₂Kit supplies the class `date` with specific display methods, along with some basic methods which allow operations like addition, comparison, or differences between dates. The class `date` is described in [Section 2.1](#).

Many applications need to send messages, to interact with the end user or to display a choice of items from which the user is to select one or several items. O₂Kit provides facilities for specifying end-user dialogue in a uniform and graphic manner: the class `box` for standardized dialogues, and the `dialog_box` for custom-designed dialogues using predefined components. Each of these classes includes several specific methods dealing with different user-interaction functions: simple messages, questions, selection, and so on. The dialogue classes are described in [Section 2.2](#).

Applications can treat bits of text as strings or collections of strings, but for longer text items it is more convenient to use a `text` class. O₂Kit provides such a class, described in [Section 2.3](#). The class `text` allows Emacs-like editing operations and contains methods to transfer text to and from Unix text files. Similarly, the `bitmap` class, described in [Section 2.4](#), allows an application to load X Window System bitmap files as O₂

objects, and provides a means to call up a standard bitmap editor to manipulate them.

The class definitions of O₂Kit are supplied in the form of a schema named `o2kit`, which is provided automatically when an O₂ user volume is initialized. In order to make use of this schema, its classes must be *imported* into the programmer's working schema. In each of the sections below, an appropriate example of the import schema command is given. Many of the facilities provided by O₂Kit make use of *specific editors*.

2.1 The class `Date`

The O₂Kit schema, supplied at system initialization time under the schema name `o2kit`, contains a class `Date` which may be imported into any other schema with the O₂ command:

```
import schema o2kit class Date
```

The structure of the class `Date` is private (refer to section Class `Date`: schema); the values of the day, month or year may be read through methods, and updates are performed through specific methods which guarantee consistency. Methods are also available to perform calculations and conversions.

Methods of class `Date`

Initialization

When an object of class `Date` is created with the O₂C primitive `new`, the programmer must supply parameters which initialize the value of each field.

```
new Date (day, month, year)
```

The class Date

creates and initializes an object of the class `Date`; `day`, `month` and `year` are integer parameters. If the year is specified as a number smaller than 100, it is interpreted as a year in the current century. Examples:

```
o2 Date dob = new Date (12, 3, 1956);  
  
o2 Date arrival = new Date (24, 6, 91);
```

If the parameters are inconsistent (such as `new Date (45, 13, -6)`) then the new `Date` object is initialized to the current date.

Reading

The three attributes `day`, `month` and `year` may be read with the following methods.

- `get_day` returns an integer representing the value of the day of month of the receiver.
- `get_month` returns an integer representing the value of the month of the receiver.
- `get_year` returns an integer representing the value of the year of the receiver. This is always the full year, including century.

Update

The three attributes `day`, `month` and `year` may be modified with the following methods which check the consistency of the `date` data input.

- `set_day (new_day)`, where `new_day` is an integer, returns a boolean (`true` if `new_day` is legal, `false` if not), and changes the value of the receiver if the modification is valid.
- `set_month (new_month)`, where `new_month` is an integer, returns a boolean (`true` if `new_month` is legal, `false` if not), and changes the value of the receiver if the modification is valid.
- `set_year (new_year)`, where `new_year` is an integer, returns a boolean (`true` if `new_year` is legal, `false` if not), and changes the value of the

receiver if the modification is valid. If `new_year` is less than 100, it is interpreted as a year in the current century.

In addition, the following method sets its receiver to the current date:

`set_to_current_date`

Addition/subtraction

The following methods make relative changes to the date encapsulated by the receiver. For subtraction, the parameter may be a negative integer.

- `add_days (days)`, where `days` is an integer, adds the specified number of days to the receiver, and returns the receiver.
- `add_months (months)`, where `months` is an integer, adds the specified number of months to the receiver, and returns the receiver.
- `add_years (years)`, where `years` is an integer, adds the specified number of years to the receiver, and returns the receiver.

Difference

The following method calculates the difference between two dates.

- `diff (another_date)`, where `another_date` is an object of class `Date`, returns a positive or negative number of days (positive if the receiver is later than the argument) representing the difference between the two dates.

Editing/display

Dates can be displayed and edited in two different formats:

- European mode (day/month/year)
- American mode (month/day/year)

The `display` and `edit` methods use a default mode, whereas the `create_presentation` method allows the programmer to specify the mode.

The class Date

Each of these methods gives a widget name (`ed_name`) to the presentation, and another to the mask created by the specific editor `date`. These widget names may be used in a resource file to specify individual graphic resources for the presentation, or for the date mask. Please refer to the *O₂Look Manual* for more details on the graphic resources for presentations.

- `display` creates and maps a non-editable presentation of the receiver; the specific editor `date` is automatically used. The widget name (`ed_name`) of the presentation is "`present_date`", and that of the date itself is simply "`date`". An example appears in [Figure 2.1](#).

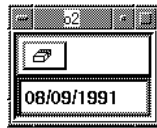


Figure 2.1: O₂Look presentation of an object of class `Date`.

- `edit` creates and maps an editable presentation of the receiver; the specific editor `date` is automatically used. Changes made by the user are validated instantly, and incorrect changes are not even echoed on the screen. The appearance of the date is the same as in [Figure 2.1](#), and the `edit` method uses the same widget names as does the `display` method.
- `create_presentation (param)` returns an O₂Look presentation identifier (a value of type `Presentation`) which can be used for editing and displaying an instance of the class `Date`. The specific editor `date` is automatically used. The single parameter `param` is an O₂ tuple value with the following structure:

```
tuple (ed_name: string,  
      mode: char)
```

The attribute `ed_name` is a string representing an identifier (a widget name) for the O₂ presentation. This identifier may be used in a resource file to set specific graphic resources for the date editor. The attribute `mode` identifies which date style to use: 'a' for American mode, 'e' for European mode.

(The widget name (`ed_name`) for the date mask itself is always `date`.)

- `get_mask`

The `get_mask` method returns a mask for the receiver object that uses the specific editor date. This is simply a shortcut for the programmer, who therefore does not need to remember the following:

```
m = lk_object ("object_date", 0, 0, lk_specific ("date", 0, 0, "date", 0,
                                                    0));
```

Note that the widget name (`ed_name`) of the date mask is `date`, while that for the object mask that encapsulates it is `object_date`. If these defaults are not suitable, or in order to set particular resources for this mask, the `lk_specific` mask may be used. (An example appears in section Using the specific editor date .)

To enter a date, the user must enter three numbers separated by any of these separators: space, slash (/), dash (-), underscore (_), or period (.). For each date element, consistency checking is done. The following examples are in European mode:

18/14/90 and 29/02/97 will be rejected as illegal dates. 15/06/1990, 15/6/90, 15 6 1990, 15-6-90, and 15-6/90 are all accepted.

Regardless of the input style used, the corresponding object will always be displayed (if the mode is European) as 15/06/1990. A graphic resource of the date editor allows displaying the years with two digits (omitting the century); the default display uses full four-digit years.

Editing a date representation involves the usual O₂Look string editing features, with the difference that no illegal date is allowed at any time. Erasing just the month, for example, is not allowed. A date may always be erased in its entirety.

Conversion to/from strings

The following methods are provided to translate a `Date` instance into a string, with the same appearance as the displayed value, or from such a string into a `Date` instance.

The class Date

- `to_string (spec)` returns a string representation of the receiver date. The single parameter `spec` is an O_2 tuple¹ value with the following structure:

```
tuple (mode: char)
```

The attribute `mode` determines the format of the string:

- `'a'` for American mode (15 June 1991 returns as "06/15/1991")
- `'e'` for European mode (15 June 1991 returns as "15/06/1991")

For example, if we are generally in American mode, the code:

```
o2 Date dd = new Date (12, 25, 1963);  
o2 string s = dd->to_string (tuple (mode: 'e'));
```

will produce the string "25/12/1963" in variable `s`.

- `to_date (spec)` returns a boolean (`true` if the specified string is legal, `false` if not), and changes the value of the receiver if the specified date string is valid. The single parameter `spec` is an O_2 tuple value with the following structure:

```
tuple (mode: char,  
       s_date: string)
```

The attributes are:

1. Frequently, O_2 Look method signatures consist of only one argument in tuple form. A single tuple argument containing multiple attributes is preferable to multiple arguments, for the following reason. The programmer might wish to define subclasses of the classes provided in O_2 Look, and to refine the behavior of these classes by supplying more information to their methods. Since the rules of signature compatibility (refer to the *O_2 C Reference manual*) forbid an inherited method from having a different number of arguments in different classes, the only way to provide additional information is with a single tuple argument containing extra attributes.

mode determines the format in which the string is to be interpreted: 'a' for American mode, 'e' for European mode.

`s_date` is the specified date string.

For example, if `dd` is an instance of class `Date`:

```
dd->to_date (tuple (mode: 'a', s_date: "9/11/1901"));
```

sets the value of `dd` to 11 September 1901.

Using the specific editor `date`

The date specific editor may be used to create an O₂Look mask for the display of date-oriented data (generally, but not necessarily, instances of the class `Date`). Date-oriented objects or values may, of course, be displayed with the usual tuple masks, but in order to make use of the special editing features and validity checking relevant to dates, the specific editor `date` must be used. The specific editor `date` is automatically used by the `display`, `edit` and `create_presentation` methods of the class `Date`. To display dates in any other context (for example, a date as an attribute of a tuple mask, or as an element in a set or list of dates), the `lk_specific` mask function must be used. Refer to the *O₂Look Reference manual* for a full explanation of this function. For the date specific editor, the `lk_specific` function must be called as follows:

```
lk_specific (ed_name, rcount, resources, date, 0, 0)
```

The arguments of the `lk_specific` mask function are as follows:

- **ed_name**: a string representing the widget name of the mask. This name may be used in a resource file to specify graphic resources for this individual mask.
- **rcount**: an integer count of the number of special resources given in the `resources` argument.

The class Date

- **resources**: a pointer to an array of graphic resources to be customized in the mask (unless **rcount** is zero). Each resource in the array is of type **Lk_resource**. For details on the specification of resources in this way, refer to the *O₂Look Manual*. A list of resources for the date editor is given below.
- **date**: the name of the specific editor mask to create.
- The two final 0 arguments indicate that the date editor does not use sub-masks.

As an example, let us assume a class **Person** containing two public attributes, **name** (a string) and **birthdate** (an instance of class **Date**). The following code creates and displays an object of class **Person**, using the date specific editor for the birthdate. While we are at it, let us change the year display so as to specify American mode.

```
o2 Person p = new Person;
Lk_presentation pid;
Lk_attribute_mask tup_mask[2];
Lk_resource res[1];
    p->name = "T. P. Guildersleeve";          /* New object of class Person */
    p->birthdate = new Date(14,4,1894);
    res[0].name = "mode";                    /* Alter year display to be      */
    res[0].value = "1";                      /* in American mode           */
    tup_mask[0].name= "name";                /* Build tuple mask for person */
    tup_mask[0].mask= lk_atom (0,0,0);        /* string for "name" and      */
    tup_mask[1].name= "birthdate";           /* lk_specific for birthdate */
    tup_mask[1].mask= lk_object (0,0,0, lk_specific(0, 1, res, "date", 0, 0));
                                              /* Use the tuple for Person object */
    pid= lk_present (p, lk_object (0,0,0, lk_tuple(0,0,0, 2, tup_mask)),
                                                              0,0,0);

    lk_map (pid, LK_COORDINATE, LK_FREE, 0, 500, 500);
    lk_wait (pid);
    lk_delete_presentation (pid);
```

The resulting presentation is shown in Figure 2.2.



Figure 2.2: A date as a tuple attribute

The class Date

Class Date: schema

```
class Date type tuple (
    day: integer,
    month: integer,
    year: integer,
    long: real)
method public init (day: integer,
                    month: integer,
                    year: integer),
    public get_day : integer,
    public get_month : integer,
    public get_year : integer,
    public set_day (day: integer): boolean,
    public set_month (month: integer): boolean,
    public set_year (year: integer): boolean,
    public set_to_current_date: Date,
    public add_days (number: integer): Date,
    public add_months (number: integer): Date,
    public add_years (number: integer): Date,
    public diff (date: Date): integer,
    public get_mask : integer,
    public menu : list(string),
    public display,
    public edit : integer,
    public create_presentation (param: tuple (ed_name: string,
                                              mode: char))
                                : integer,
    public to_string (spec: tuple(mode: char)) : string,
    public to_date (new_date: tuple(mode: char,
                                    s_date: string))
                    : boolean,
    private in_a_leap_year : boolean,
    private compute_long,
    private days_to_date (days: real),
    private in_a_good_day: boolean
end;
function year_size(year: integer) : integer;
function in_a_good_day(day: integer, month : integer, year: integer) :
boolean;
function is_separator(c: char ) : boolean;
function estimated_day(day: integer, month: integer, year: integer) :
integer;
export schema class Date;
```

2.2 Dialogue boxes

If the programmer needs to display a message or ask the end user for input or for a selection from a list of items, the dialogue boxes provide two kinds of tools:

The first kind is very easy to use: It encapsulates the parameters of O₂Look in a class named `Box`. This class has very simple methods for constructing a dialogue with the end user (message, question, selection).

The second kind is much more sophisticated; it allows the programmer to build custom-designed dialogue boxes from components, each provided with its specific display methods. The result is a presentation which can be used by the O₂Look functions.

The class `Box`

The class `Box` is sufficient for the specification of simple dialogues. It has predefined methods where the O₂Look parameters are hidden from the programmer. Each method will display a presentation, and erase it when a button of the presentation is pressed. The execution of the application is suspended until the user clicks on one of the buttons. During the display of the dialogue box, the user is allowed to manipulate any other presentation on the screen. The methods provide two means of specifying dialogue strings. The simpler way is through a string parameter. However, if this parameter is specified as the null string, the dialogue string is taken instead from an O₂Look graphic resource, `labelString`. This graphic resource is assigned, in a resource file, to a particular dialogue box according to the widget name of that box; the programmer supplies the widget name in a parameter called `ed_name` of the method. In this way, context-sensitive messages (for example, depending upon what language the end user speaks) can be generated. For specifying graphic resources for a dialogue box in a resource file, the name used is:

```
app_name.dialog.ed_name.res_name: value
```

or

Dialogue boxes

```
app_name*ed_name.res_name: value
```

where

- `app_name` — the application name (see `lk_prologue`)
- `ed_name` — the string pass to `Dialoguer` methods
- `value` — the value of the resources.

Note that `dialog` is a set word and is obligatory.

Methods of class `Box`

To use the methods of the class `Box`, you import the class `Box`, then immediately declare and initialize a named object of that class. This object serves as an all-purpose receiver for the useful `Box` methods. In all of the examples below, the `Box` object is called `Dialoguer`.

```
import schema o2kit class Box;  
name Dialoguer: Box;  
run body {Dialoguer = new Box;};
```

The methods that can be invoked on `Dialoguer` are as follows:

```
message (label: string, ed_name: string)
```

This method displays a message in a dialogue box, waits for the user to click on the “OK” or “Cancel” button, and returns nothing to the calling program. Parameters:

- `label` — the string to be displayed; if this is empty, the message is taken from the resource `labelstring`.

- **ed_name** — the widget name of the dialogue box. This name may be used in a resource file to specify graphic resources for this individual dialogue box.

Example (Figure 2.3):

```
Dialoguer->message ("This name is already used! You must  
use another one", "")
```

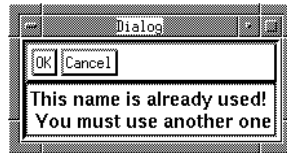


Figure 2.3: A message dialogue.

```
question (label: string, ed_name: string): boolean
```

This method displays a yes/no question in a dialogue box, waits for the user to click on the “Yes” or “No” button, and returns a boolean value to the calling program (true if “Yes” was clicked, false if “No”). Parameters:

- **label** — the string to be displayed; if this is empty, the question is taken from the resource `labelString`.
- **ed_name** — the widget name of the dialogue box. This name may be used in a resource file to specify graphic resources for this individual dialogue box.

Example (Figure 2.4):

```
Dialoguer->question ("Do you want to save your changes?",  
"")
```

Dialogue boxes

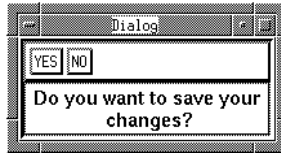


Figure 2.4: A question dialogue.

```
dialog (label: string, ed_name: string): string
```

This method is useful if the application requires a string as an answer to a question. The method will display the question, along with a text edit area. If the user clicks on the “OK” button, the method returns the string which is in the text edit area; if the user clicks on the “Cancel” button, the method returns the empty string. Parameters:

- **label** — the string to be displayed.
- **ed_name** — the widget name of the dialogue box. This name may be used in a resource file to specify graphic resources for this individual dialogue box.

Example (Figure 2.5):

```
Dialoguer->dialog ("Enter the name", "")
```



Figure 2.5: A question-and-answer dialogue.

```
selection (title: string, ed_name: string, items: list(string)): string
```

This method allows the user to select one string from a list, initialized by the programmer. The selection is made by clicking the left mouse button. The list appears in a scrolling window if the number of items is greater than a fixed number (four by default). The user must confirm the selection by clicking on the “OK” button. If the user doesn’t want to select any of the strings, the “Cancel” button may be clicked. The method returns the selected string if the “OK” button was pressed; if the “Cancel” button was pressed, the method returns the empty string. Parameters:

- **title** — the title of the selection box.
- **ed_name** — the widget name of the dialogue box. This name may be used in a resource file to specify graphic resources for this individual dialogue box.
- **items** — the list of strings from which the user is to choose. Its type must be list(string).

Example (Figure 2.6):

```
Dialoguer->selection ("Select one item", "",  
list("first", "second", "third", "fourth", "fifth"))
```

If the user were to click the “OK” button in Figure 2.6, the method would return the string "third".



Figure 2.6: A simple selection dialogue.

Dialogue boxes

```
edit_selection (title: string, ed_name: string, items: list(string)): string
```

With this selection box the end user may either select a string in the list, or enter a new string in a text edit area. If a string is selected, it appears automatically in the text edit area of the box, and may be altered. If the “OK” button is pressed, the method returns the string which is displayed in the edit area; if the “Cancel” button is pressed, the method returns the empty string. Parameters:

- **title** — the title of the selection box.
- **ed_name** — the widget name of the dialogue box. This name may be used in a resource file to specify graphic resources for this individual dialogue box.
- **items** — the list of strings from which the user may choose. Its type must be `list(string)`.

Example (Figure 1.7):

```
Dialoguer$\rightarrow$edit_selection ("Choose one name",  
"", list("first", "second", "third", "fourth", "fifth"))
```

If the user were to click the “OK” button in Figure 2.7, the method would return the string “third”.



Figure 2.7: An editable selection dialogue.

```
mult_selection (title: string, ed_name: string,  
               items: list(string)): list(string)
```

The user can choose one or several strings from a list; the method returns the list of the selected strings if the “OK” button is pressed, or the empty string if the “Cancel” button is pressed. Parameters:

- **title** — the title of the selection box.
- **ed_name** — the widget name of the dialogue box. This name may be used in a resource file to specify graphic resources for this individual dialogue box.
- **items** — the list of strings from which the user may choose. Its type must be `list(string)`.

Example (Figure 2.8):

```
Dialoguer->mult_selection ("Select several items", "",  
                           list("first", "second", "third", "fourth", "fifth"))
```

If the user were to click the “OK” button in Figure 2.8, the method would return the value:

```
list ("third", "fifth")
```

Dialogue boxes



Figure 2.8: A multiple-selection dialogue.

The class `Component`

Objects of the class `Component` are “building-block” dialogue boxes which may be used directly by the programmer for the construction of customized dialogue boxes. Each different type of atomic dialogue box (radio box, selection box, message, etc.) is given its own subclass of the class `Component`. Instances of the subclasses of `Component` are generally (but not necessarily) used within the framework of an object of class `Dialog_box` (in section The class `Dialog_box`), which englobes them. `O2Look` provides each of the subclasses with local definitions of three methods:

`init`

This creates and initializes a new object of the class (and is invoked by the `O2C` instruction `new` — refer to the *O₂C Reference manual*) The method requires two kinds of parameters: one for a widget name (needed only for the specification of `O2Look` resources in a resource file), and one or more for the initialization of the content of the dialogue box.

`create_presentation`: integer

Builds a specific presentation of the object, and returns it as a presentation identifier (a value of type `Lk_presentation`). This method is useful only when the object is to be displayed alone, and not as a part of a `Dialog_box` object.

get_mask

Returns a specific editor mask (a value of type `Lk_mask`) which can be used by the O₂Look primitives. This method is used by the `create_presentation` method (above), and is indispensable to the construction of a complex dialogue box. Refer to Section [Ref: pbbex] for a programming example.

Where relevant, a method called `answer` is also provided, whose returned value indicates the selection made by the end user; this method should be invoked only after the O₂Look function `lk_consult`. All of the O₂Look functions and methods can be used on objects of these classes. The classes provided are:

Button

Instances of class `Button` are components of instances of classes `Button_box` and `Radio_box`. The `init` method takes two parameters, and is invoked as:

```
new Button (ed_name, label)
```

where `ed_name` is a string representing the widget name used for resource specification, and `label` is the string that appears to the right of the button (Figures 2.9 and 2.10). If `label` is the null string, the label is taken from the graphic resource `labelstring`. The `label` argument is ignored entirely if the `labelType` resource is set to `PIXMAP` instead of the default `STRING`. This allows the use of bitmap or pixmap images instead of text strings to identify buttons.

The `Button` class lacks a `create_presentation` method, because objects of this class are never displayed alone; they are always components of `Radio_box` or `Button_box` objects.

Dialogue boxes

Radio_box

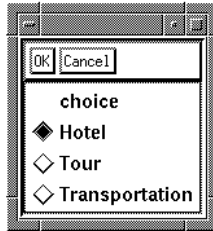


Figure 2.9: A radio box allows only one button selection.

A radio box component displays buttons with either text or bitmaps to their right. (The `O2Look` resource `labelType` allows this choice for each button.) Buttons are selected with the left mouse button. Two buttons cannot be selected together; the selection of one button automatically cancels any previous selection.

The `init` method takes four parameters, and is invoked as:

```
new Radio_box (ed_name, title, buttons, selected)
```

where:

- `ed_name` is a string representing the widget name used for resource specification.
- `title` is the title string of the radio box component (the word “Choice” in Figure 2.9).
- `buttons` (a value of type `list(Button)`) is the list of component buttons of the radio box.
- `selected` is an integer showing which button is to be pre-selected. Remember that the first button is numbered zero.

The method `answer` returns the ordinal number of the selected button (the first is numbered zero). In Figure 2.9, this would be the integer 0.

Button_box

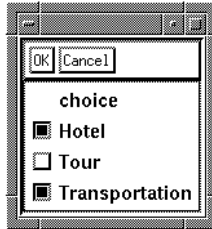


Figure 2.10: A button box allows multiple selections.

A button box component displays buttons with either text or bitmaps to their right. (The O₂Look resource `labelType` allows this choice for each button.) Buttons are selected and deselected with the left mouse button; clicking on a button reverses its state. The user may select as many buttons as needed.

The `init` method takes four parameters, and is invoked as:

```
new Button_box (ed_name, title, buttons, selected)
```

where:

- `ed_name` is a string representing the widget name used for resource specification.
- `title` is the title string of the button box component (the word “Choice” in Figure 2.10).
- `buttons` (a value of type `list(Button)`) is the list of component buttons of the button box.
- `selected` is a list of integers showing which buttons are to be pre-selected. Remember that the first button is numbered zero.

The method `answer` returns a list of integers representing the ordinal numbers of the selected buttons (the first is numbered zero). In Figure 2.10, this would be `list(0,2)`.

Dialogue boxes

Single_selection

A **single-selection** component displays a list of strings, as seen in Figure . A string is selected with the left mouse button. Only one item can be selected; selecting a second string automatically deselects the first one.

The **init** method takes four parameters, and is invoked as:

```
new Single_selection (ed_name, title, items, selected)
```

where:

- **ed_name** is a string representing the widget name used for resource specification.
- **title** is the title string of the selection box component (the phrase “Select one item” in Figure 2.6).
- **items** is the list of strings from which the user is to choose.
- **selected** is an integer showing which string item is to be pre-selected. Remember that the first string is numbered zero.

The method **answer** returns the ordinal number of the selected string (the first is numbered zero). In Figure 2.6, this would be the integer 2.

Editable_selection

An **editable-selection** component displays a list of strings with a text edit area, as seen in Figure 2.7. A string may be selected with the left mouse button. Only one item can be selected; selecting a second string automatically deselects the first one. The selected string appears in the text edit area. The user may change the contents of this area at any time, by typing into it.

The **init** method takes five parameters, and is invoked as:

```
new Editable_selection (ed_name, title, items, selected, other)
```

where:

- **ed_name** is a string representing the widget name used for resource specification.
- **title** is the title string of the selection box component (the phrase “Choose one name” in Figure 2.7).
- **items** is the list of strings from which the user is to choose.
- **selected** is an integer showing which string item is to be pre-selected. Remember that the first string is numbered zero.
- **other** is another string used to initialize the text edit area. If this string is null, the text edit area is initialized with the pre-selected string indicated by the **selected** argument. (This means that in order to initialize the text edit area to blanks, **other** must be a blank string and not a null one — that is, " " and not "").)

The method **answer** returns the string appearing in the text edit area. In Figure 2.7, this would be the string “third”.

Multiple_selection

A multiple-selection component displays a list of strings, as seen in Figure 2.8. A string is selected and deselected with the left mouse button; clicking on a string changes its selection state. The user may select as many strings as needed.

The **init** method takes four parameters, and is invoked as:

```
new Multiple_selection (ed_name, title, items, selected)
```

where:

Dialogue boxes

- **ed_name** is a string representing the widget name used for resource specification.
- **title** is the title string of the selection box component (the phrase “Select several items” in Figure 2.8).
- **items** is the list of strings from which the user is to choose.
- **selected** is a list of integers showing which string items are to be pre-selected. Remember that the first string is numbered zero.

The method **answer** returns a list of integers representing the ordinal numbers of the selected strings (the first is numbered zero). In Figure 2.8, this would be `list(2,4)`.

Prompt

A prompt component displays a question and a text edit area for the answer. The end user types the answer into the text edit area.

The **init** method takes three parameters, and is invoked as:

```
new Prompt (ed_name, title, answer)
```

where:

- **ed_name** is a string representing the widget name used for resource specification.
- **title** is the prompt string (question).
- **answer** is a string used to initialize the contents of the text edit area.

The method **answer** returns the string appearing in the text edit area.

Label

A label component simply displays a message; it can be used, for example, to set the title of a dialogue box. It has no answer method.

The `init` method takes two parameters, and is invoked as:

```
new Label (ed_name, title)
```

where:

- `ed_name` is a string representing the widget name used for resource specification.
- `title` is the message string to display.

In addition to the usual `create_presentation` method, the class `Label` includes another method called `create_presentation_question`; this performs exactly the same way as `create_presentation` except that the presentation's Pencil button is replaced with the word "Yes" and its Eraser button with the word "No".

Picture

A picture component allows the placement of a bitmap or pixmap image in a dialogue box. It has no answer method. Its main purpose is to supply a widget name, for which the O₂Look graphic resource `labelPixmap` may be established to identify the bitmap or pixmap file containing the image. (An example appears in Section [Two examples](#).)

The `init` method takes only one parameter:

```
new Picture (ed_name)
```

where `ed_name` is a string representing the widget name used for resource specification.

Dialogue boxes

These subclasses are detailed in the dialogue schema given in Section [Dialogue boxes: schema](#); examples of their use appear in Section [Two examples](#). A schema diagram of all the dialogue box classes is shown in Figure 2.12. The programmer is free to add new classes to this list. They should be defined as subclasses of the class `Component`, and include a method named `get_mask` which creates and returns a mask that displays the component. New subclasses of `Component` should probably also define their own `init` methods and, if relevant, answer methods. In order to use the definitions of these classes, the programmer must import them from the `o2kit` schema into his/her own schema. Only those classes actually used need to be imported. For example, to build button boxes and radio boxes within the framework of an object of class `Dialog_box` (Section The class `Dialog_box`), the following `import` command would suffice:

```
import schema o2kit class Button, Button_box, Radio_box, Dialog_box
```

To add new `Component` subclasses, the class `Component` should be imported as well.

The class `Dialog_box`

The programmer may use instances of the class `Dialog_box` to build customized dialogue boxes with objects of some or all the subclasses of the class `Component`. An instance of the class `Dialog_box` is structured as a list of rows, each row being itself a list of components. Its structure is thus a list of lists of elements of the class `Component`. To use the class `Dialog_box` in a schema, it must be imported from the schema `o2kit`. An example of this appears at the end of Section The class `Component`. The methods of the class `Dialog_box` are as follows:

`init`

creates and initializes a new object of the class `Dialog_box`. Two parameters are required:

```
new Dialog_box (ed_name, structure)
```

where:

- `ed_name` is a string representing the widget name used for resource specification.
- `structure` is a value of type list (`list (Component)`), as described above.

```
create_presentation: integer
```

builds a specific presentation of the dialogue box, and returns it as a presentation identifier (a value of type `Lk_presentation`).

```
element (i: integer, j: integer): Component
```

returns the component from the `i`th row and the `j`th column (each beginning with 0) of the dialogue box. If there is no component at this position, the method returns a “nil” object.

All O₂Look functions and methods may be used on these objects. In fact, the methods of the class `Box` (Section [The class Box](#) is all written using instances of the class `Dialog_box` and its methods. Section [Two examples](#) gives an example of this implementation.

Two examples

Two examples are presented here: a simple one and a more complicated one. In each case, a taste of the internal implementation of O₂Look is given as well, in the form of some of the O₂Look methods.

Dialogue boxes

A simple dialogue box

The following code:

```
Dialoguer->selection ("Select one item", "",  
    list("first", "second", "third", "fourth", "fifth"))
```

would produce this dialogue box:



The named object `Dialoguer` belongs to class `Box`. Here is the code of the method `selection` of the class `Box`:

```
method body selection (title: string, ed_name: string, items:  
list(string)): string  
    in class Box {  
        o2 Single_selection lb;  
        int n, i;  
        int pid;  
  
        lb = new Single_selection ("choice", title, items, 0);  
        /* The ed_name "choice" can be used to set resources for the dialogue box */  
        pid = lb->create_presentation;  
        lk_map ((Lk_presentation)pid, LK_MOUSE, LK_FREE, 0, 0, 0);  
        n = lk_wait ((Lk_presentation)pid);  
        lk_consult (pid, lb);  
        i = lb->answer;  
        lk_delete_presentation ((Lk_presentation) pid);  
        if (n == LK_SAVE) return(items[i]); else return("");  
    };
```

The method `create_presentation` in the class `Single_selection` is implemented as follows:

```
method body create_presentation: integer in class Single_selection {
    int pid;
    o2 string str;
    lk_resource res[8];
    res[0].name = "penType";
    res[0].value = "STRING";
    res[1].name = "penString";
    res[1].value = "OK";
    res[2].name = "lockedPenType";
    res[2].value = "STRING";
    res[3].name = "lockedPenString";
    res[3].value = "OK";
    res[4].name = "eraserType";
    res[4].value = "STRING";
    res[5].name = "eraserString";
    res[5].value = "Cancel";
    res[6].name = "lockedEraserType";
    res[6].value = "STRING";
    res[7].name = "lockedEraserString";
    res[7].value = "Cancel";
    str = self->ed_name;
    pid = lk_present (self, lk_specific(str, 0, 0, "ssel", 0, 0), "dialog", 8,
    res);
    return (pid);
};
```

Note that the last `lk_present` function uses a specific editor mask called `"ssel"`. The behavior of each dialogue box component is governed by specific editors. The named object `Dialoguer` of the class `Box` is used to send the messages.

```
o2 string str;
str = Dialoguer->selection ("Select one item", "", list ("first", "second",
    "third", "fourth", "fifth"));
```

Dialogue boxes

A customized dialogue box

The next example is more complicated, so the programmer cannot use the class `Box` or the `Dialoguer` object. The following dialogue box is required:

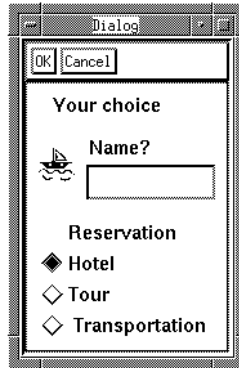


Figure 2.11: A complex dialogue box.

The programmer must code the structure and all of the display parameters.

```

o2 Dialog_box dialog;
o2 Picture I;
o2 Label lb;
o2 Prompt pr;
o2 Button b1, b2, b3;
o2 Radio_box rd;
o2 string name;
o2 list (Component) l1, l2, l3;
int pid, n, choice ;
I = new Picture ("bitmap1");          /* for labelPixmap resource */
lb = new Label ("label1", "Your choice"); /* the title of the box */
pr = new Prompt ("prompt1", "Name?", " "); /* a question with a text edit
area*/
b1 = new Button ("b1", "Hotel");
b2 = new Button ("b2", "Tour");
b3 = new Button ("b3", "Transportation");
rd = new Radio_box ("radio1", "Reservation", list(b1, b2, b3), 0);
/* a radio box where the title is ``Reservation``, the strings */
/* attached to the buttons are ``Hotel``, ``Tour``, ``Transportation``, */
/* and the first button is preselected. */
l1 = list(lb);
l2 = list((o2 Component) I, pr);
l3 = list(rd);
dialog = new Dialog_box ("dialog1", list (l1, l2, l3));
/* first line (l1): the title */
/* second line (l2): the bitmap, followed by the prompt */
/* third line (l3): the radio box */
pid = dialog->create_presentation;
lk_map (pid, LK_MOUSE, LK_FREE, 0, 0, 0);
n = lk_grab (pid);
lk_consult (pid, dialog);
if (n == LK_SAVE) {
    name = pr->answer;          /* access to the prompt */
    choice = rd->answer;        /* access to the radio box */
/* Note that these last two lines could have been:
    name = ((o2 Prompt)(dialog->element (1,1)))->answer;
    choice = ((o2 Radio_box)(dialog->element (2,0)))->answer;*/
}
lk_unmap (pid);

```

In a resource file, a line such as the following would identify the bitmap for the sailboat image:

```
*dialog1*bitmap1.labelPixmap :    /usr/share/X11/bitmaps/sailboat.bitmap
```

Dialogue boxes

The method `create_presentation` of class `Dialog_box` is implemented as follows:

```
method body create_presentation: integer in class Dialog_box {
  Lk_resource res[8];
  o2 string str;
  o2 integer pid;
  Lk_mask masks;
  res[0].name = "penType";
  res[0].value = "STRING";
  res[1].name = "penString";
  res[1].value = "OK";
  res[2].name = "lockedPenType";
  res[2].value = "STRING";
  res[3].name = "lockedPenString";
  res[3].value = "OK";
  res[4].name = "eraserType";
  res[4].value = "STRING";
  res[5].name = "eraserString";
  res[5].value = "Cancel";
  res[6].name = "lockedEraserType";
  res[6].value = "STRING";
  res[7].name = "lockedEraserString";
  res[7].value = "Cancel";
  masks = lk_method ("get_mask");
  str = self->ed_name;
  pid = lk_present (self, lk_specific(str, 0, 0, "dialog", 1 ,&masks),
                   "dialog", 8, res);

  return (pid);
};
```

The `create_presentation` method begins by redefining the aspect of the two presentation buttons (ordinarily a “Pencil” and an “Eraser” button) to show the strings “OK” and “Cancel”. Then it builds the presentation of the `Dialog_box` object, using its `get_mask` method to define the mask used to present the object. The final `lk_present` function invokes the specific editor named “dialog”, supplying it with the submask returned by `get_mask`. Herein lies the importance of the `get_mask` function: *all* presentations of objects of class `Dialog_box` use a specific editor that requires submasks, and `get_mask`, called by the mask function `lk_method`, is the only way to furnish them.

Graphic resources for dialogue boxes

The following tables list the graphic resources which may be customized for dialogue boxes. The customizations appear in a resource file, in association with the *ed_name* argument supplied when the dialogue object was created with the `O2C` instruction `new`. Refer to the *O₂Look Manual* for more information on the specification of graphic resources in resource files.

Button resources

The following resources are available for button displays (that is, instances of class `Button`).

Name	Type	Default	Access
<code>foregroundColor</code>	Color	Dynamic	CS
<code>backgroundColor</code>	Color	Dynamic	CS
<code>fontList</code>	string	"fixed"	CS
<code>labelType</code>	unsigned char	STRING	CS
<code>labelPixmap</code>	string	UNSPECIFIED_PIXMAP	CS
<code>labelString</code>	string	**	CS

- **`foregroundColor`**

Specifies the foreground color for the button.

- **`backgroundColor`**

Specifies the background color for the button.

- **`fontList`**

Specifies the font used for the button label.

- **`labelType`**

Specifies whether the button label appears in the form of a string (STRING) or a bitmap or pixmap image (PIXMAP).

- **`labelPixmap`**

Dialogue boxes

Specifies the name of the bitmap or pixmap file to display if `labelType` is `PIXMAP`.

- **labelString**

Specifies the string to display if `labelType` is `STRING`. Note that the *label* argument given when a `Button` instance is created will override this resource specification, unless the *label* argument is a null string.

Button box and radio box resources

The following resources are available for instances of the classes `Button_box` and `Radio_box`.

Name	Type	Default	Access
<code>foregroundColor</code>	Color	Dynamic	CS
<code>backgroundColor</code>	Color	Dynamic	CS
<code>titleFontList</code>	string	"fixed"	CS
<code>orientation</code>	unsigned char	VERTICAL	CS

- **foregroundColor**

Specifies the foreground color for the box area.

- **backgroundColor**

Specifies the background color for the box area.

- **titleFontList**

Specifies the font used for the box title.

- **orientation**

Specifies whether the buttons are placed vertically (`VERTICAL`) or horizontally (`HORIZONTAL`).

Single selection and multiple selection resources

The following resources are available for single selection and multiple selection dialogues (that is, instances of the classes *Single_selection* and *Multiple_selection*).

Name	Type	Default	Access
foregroundColor	Color	Dynamic	CS
backgroundColor	Color	Dynamic	CS
titleFontList	string	"fixed"	CS
fontList	string	"fixed"	CS
visibleItemCount	short	4	CS
visibleItemCountStatic	Boolean	False	CS

- **foregroundColor**

Specifies the foreground color for the selection box.

- **backgroundColor**

Specifies the background color for the selection box.

- **titleFontList**

Specifies the font used for the title.

- **fontList**

Specifies the font used for the selection strings.

- **visibleItemCount**

Specifies the number of selection strings visible at one time. If there are more than this number of them, a scroll bar appears; if there are fewer, the **visibleItemCountStatic** resource comes into play.

- **visibleItemCountStatic**

Determines whether or not the selection box always has the same size. If **True**, the box always has room for the number of selection strings specified in the **visibleItemCount** resource, even if there are not that many strings available. If **False** (the default), the size of the box will vary according to the number of strings available (as long as this is less than or equal to **visibleItemCount**).

Dialogue boxes

Editable selection resources

The following resources are available for editable selection dialogues (that is, instances of the class `Editable_selection`).

Name	Type	Default	Access
<code>foregroundColor</code>	Color	Dynamic	CS
<code>backgroundColor</code>	Color	Dynamic	CS
<code>titleFontList</code>	string	"fixed"	CS
<code>fontList</code>	string	"fixed"	CS
<code>visibleItemCount</code>	short	4	CS
<code>visibleItemCountStatic</code>	Boolean	False	CS
<code>columns</code>	short	10	CS

- **`foregroundColor`**

Specifies the foreground color for the selection box.

- **`backgroundColor`**

Specifies the background color for the selection box.

- **`titleFontList`**

Specifies the font used for the title.

- **`fontList`**

Specifies the font used for the selection strings and the text edit area.

- **`visibleItemCount`**

Specifies the number of selection strings visible at one time. If there are more than this number of them, a scroll bar appears; if there are fewer, the `visibleItemCountStatic` resource comes into play.

- **`visibleItemCountStatic`**

Determines whether or not the selection box always has the same size. If `True`, the box always has room for the number of selection strings

specified in the `visibleItemCount` resource, even if there are not that many strings available. If `False` (the default), the size of the box will vary according to the number of strings available (as long as this is less than or equal to `visibleItemCount`).

- **`columns`**

Specifies the width of the text edit area, in characters.

Prompt resources

The following resources are available for displays of objects of class `Prompt`.

Name	Type	Default	Access
<code>foregroundColor</code>	Color	Dynamic	CS
<code>backgroundColor</code>	Color	Dynamic	CS
<code>titleFontList</code>	string	"fixed"	CS
<code>fontList</code>	string	"fixed"	CS
<code>columns</code>	short	10	CS

- **`foregroundColor`**

Specifies the foreground color for the prompt.

- **`backgroundColor`**

Specifies the background color for the prompt.

- **`titleFontList`**

Specifies the font used for the prompt (the question).

- **`fontList`**

Specifies the font used for the text edit area (the answer).

- **`columns`**

Specifies the width of the text edit area, in characters.

Dialogue boxes

Label resources

The following resources are available for displays of instances of class `Label`.

Name	Type	Default	Access
<code>foregroundColor</code>	Color	Dynamic	CS
<code>backgroundColor</code>	Color	Dynamic	CS
<code>fontList</code>	string	"fixed"	CS
<code>labelString</code>	string	""	CS

- **`foregroundColor`**

Specifies the foreground color for the label message.

- **`backgroundColor`**

Specifies the background color for the label message.

- **`fontList`**

Specifies the font used for the label message.

- **`labelString`**

Specifies the label message to display. Note that this resource specification may be overridden by giving a non-null *title* argument when creating a new instance of the class `Label`.

Picture resources

The following resources are available for displays of instances of class `Picture`.

Name	Type	Default	Access
<code>foregroundColor</code>	Color	Dynamic	CS
<code>backgroundColor</code>	Color	Dynamic	CS
<code>labelPixmap</code>	string	UNSPECIFIED_PIXMAP	CS

- `foregroundColor`

Specifies the foreground color for the picture.

- `backgroundColor`

Specifies the background color for the picture.

- `labelPixmap`

Specifies the name of the bitmap or pixmap file to display. Note that specifying this resource is the *only* way to indicate which bitmap or pixmap to display.

Dialogue boxes: schema

```
class Dialog_box type tuple (ed_name: string,
                             structure : list (list (Component))
                             )
method public init (ed_name: string, structure: list(list(Component))),
                  public create_presentation: integer,
                  public element(i:integer, j:integer): Component
end
export schema class Dialog_box;
class Component type tuple (ed_name: string
                             )
method          public get_mask : integer,
                  public create_presentation : integer
end;
export schema class Component;
class Button    type tuple (ed_name: string,
                             label: string
                             )
method public init (ed_name: string, label: string),
                  public get_mask : integer
end;
export schema class Button;
class Label inherit Component type tuple (
                                     label:string
                                     )
```

Dialogue boxes

```
method public init (ed_name: string, label: string),
    public get_mask : integer,
    public create_presentation : integer,
    public create_presentation_question : integer
end;
export schema class Label;
class Picture inherit Label
method public init (ed_name: string),
    public create_presentation : integer,
    public get_mask : integer
end;
export schema class Picture;
class Prompt inherit Component type tuple (
    title : string,
    reply : string
)
method public init (ed_name: string, title: string, answer: string),
    public answer: string,
    public get_mask : integer,
    public create_presentation : integer
end;
export schema class Prompt;
class Single_selection inherit Component type tuple (
    title           : string,
    items           : list(string),
    selected        : integer
)
method public init(ed_name           : string,
    title           : string,
    items           : list(string),
    selected        : integer),
    public get_mask : integer,
    public answer: integer,
    public create_presentation : integer
end;
export schema class Single_selection;
class Editable_selection inherit Component type tuple (
    title           : string,
    items           : list (string),
    selected        : integer,
    other           : string
)
method public init(ed_name           : string,
    title           : string,
    items           : list (string),
    selected        : integer,
    other           : string ),
```

```

        public get_mask : integer,
        public answer: string,
        public create_presentation : integer
end;
export schema class Editable_selection;
class Multiple_selection inherit Component type tuple (
        title      : string,
        items      : list (string),
        selected    : list (integer)
)
method public init(ed_name : string,
        title      : string,
        items      : list (string),
        selected    : list (integer)),
        public get_mask : integer,
        public answer: list(integer),
        public create_presentation : integer
end;
export schema class Multiple_selection;
class Button_box inherit Component type tuple (
        title      : string,
        buttons     : list (Button),
        selected    : list (integer)
)
method public init(ed_name : string,
        title      : string,
        buttons     : list (Button),
        selected: list (integer)),
        public get_mask : integer,
        public answer: list (integer),
        public create_presentation : integer
end;
export schema class Button_box;
class Radio_box inherit Component type tuple (
        title      : string,
        buttons     : list (Button),
        selected    : integer
)
method public init(ed_name      : string,
        title      : string,
        buttons     : list (Button),
        selected    : integer),
        public answer: integer,
        public get_mask : integer,
        public create_presentation : integer
end;

```

Dialogue boxes

```
export schema class Radio_box;
class Box
method  public message (label: string, ed_name: string),
        public question(label: string, ed_name: string): boolean,
        public dialog (label: string, ed_name: string) : string,
        public selection (title: string, ed_name: string, items:
                           list(string)): string,
        public edit_selection (title: string, ed_name: string, items:
                               list(string)): string,
        public mult_selection (title: string, ed_name: string, items:
                                list(string)): list(string)
end;
export schema class Box;
```

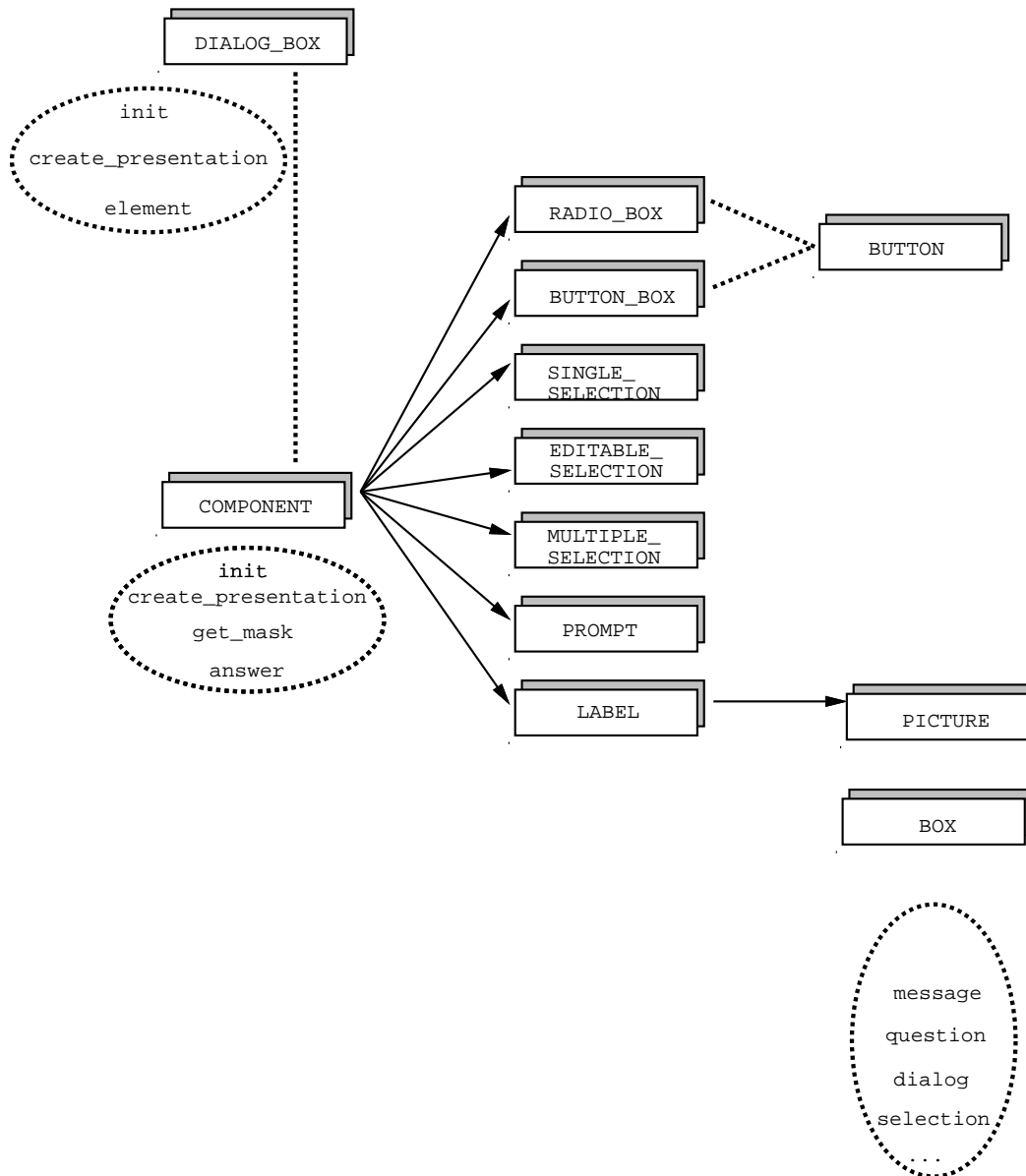


Figure 2.12: Dialogue box schema.

2.3 The class `Text`

The O₂Look schema provides a class `Text` which may be imported into any other schema with the O₂ command:

The class Text

```
import schema o2kit class Text
```

The structure of the class `Text` is a one-attribute tuple² containing a list of strings. The `display` and `edit` masks attached to this class make use of the specific editor `text`, which allows full-screen text editing operations with Emacs-like control commands.

Methods of the class `Text`

`read_file (filename, mode)`

This method reads a Unix text file and loads its contents into the value of the receiver object. The parameters are:

- **filename**: a string containing the full Unix path and name of the file to be read.
- **mode**: a string containing either:
 - * `"a"` (append) — The contents of the Unix file are appended to any text already contained in the value of the receiver object.
 - * `"w"` (write) — The value of the receiver object is initialized with the contents of the Unix file; any previous value of the object is lost.

The appropriate error message is generated if the file is not found, or if *mode* was not `"a"` or `"w"`.)

`write_file (filename, mode)`

This method takes the receiver object value and writes it to a Unix file. The parameters are:

- **filename**: a string containing the full Unix path and name of the file to be written.

2. One-attribute tuples have the advantage that they may be refined in user-defined subclasses as multiple-attribute tuples. This would not be possible if, for example, the type structure of the class `Text` had been simply `list (string)`.

- `mode`: a string containing either:

- * `"a"` (append) — The value of the receiver object is appended to the Unix file.

- * `"w"` (write) — The specified Unix file is created (removing any existing file of the same name), and the value of the receiver object is written to that file.

The method generates an appropriate error message if the file could not be created or was not found, or if *mode* was neither `"a"` nor `"w"`.)

`display`

The generic method `display` is redefined to use the specific editor `text`. The presentation is not editable.

`edit`

The generic method `edit` is redefined to use the specific editor `text`. The presentation is editable.

`get_mask`

The `get_mask` method returns a mask for the receiver object that uses the specific editor `text`. This is a shortcut so you do not have to remember the following:

```
m = lk_object ("object_text", 0, 0, lk_specific ("text", 0, 0, "text", 0, 0));
```

The widget name (*ed_name*) of the text mask is `"text"`. For the object mask that encapsulates it, the name is `"object_text"`. Use the `lk_specific` mask to set other resources.

An example

The following O₂C code:

The class Text

```
o2 Text text = new Text;
text->read_file ("/u/mark/text/grace.txt", "w"); /* loads UNIX file */
if (text->edit == LK_SAVE) {
    text->write_file("/u/mark/text/grace.txt", "w"); /* updates UNIX file */
}
```

reads a Unix text file, displays it on the screen as in Figure 2.13, allows the text to be edited, and then saves any changes back to the Unix file.

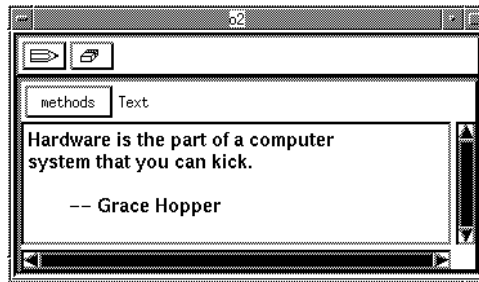


Figure 2.13: An object of class Text.

For those programmers who feast on detail, the effect of the above code may be duplicated using the `get_mask` method as follows:

```
o2 Text text = new Text;
Lk_presentation pid;
Lk_mask m;
text->read_file ("/u/mark/text/grace.txt", "w"); /* loads UNIX file */
m = text->get_mask;
pid = lk_present (text, m, "present_text", 0, 0);
lk_map (pid, LK_MANUAL, LK_FREE, LK_SCREEN, 0, 0);
if (lk_wait (pid) == LK_SAVE) {
    lk_consult (pid, text);
    text->write_file("/u/mark/text/grace.txt", "w"); /* updates UNIX file */
}
lk_delete_presentation (pid);
```

Class `Text`: schema

```
class Text public
  type
    tuple (content: list (string))
  method
    error_msg (label: string),
    public read_file (filename: string, mode: string): integer,
    public write_file (filename: string, mode: string): integer,
    public display,
    public edit: integer,
    public get_mask: integer
end;
export schema class Text;
```

2.4 The class `Bitmap`

The O₂Look schema has a class `Bitmap` which you import into another schema using the O₂ command:

```
import schema o2kit class Bitmap
```

The structure of the class `Bitmap` is a tuple reflecting the standard bitmap format of the X Window System:

```
tuple (width: integer,
      height: integer,
      bitmap: bits)
```

The `display` method attached to this class uses the specific editor `bitmap`, which presents the bitmap as a graphic image rather than a tuple. Clicking the right mouse button on the image calls up a standard bitmap editor.

The class `Bitmap`

Methods of class `Bitmap`

`loadfile (filename)`

Reads a Unix file containing bitmap data in X Window System format, and initializes the receiver object value with the bitmap data read.

- **filename**: a string containing the full Unix path and name of the bitmap file to be read.

The method generates an appropriate error message if the file was not found, or if it did not contain bitmap data in the expected format.

`display`

The generic method `display` is redefined to use the specific editor `bitmap`. The presentation is editable (by clicking the right mouse button on the image).

`get_mask`

The `get_mask` method returns a mask for the receiver object that uses the specific editor `bitmap`. This is simply a shortcut for the programmer, who therefore does not need to remember the following:

```
m = lk_object ("object_bitmap", 0, 0, lk_specific ("bitmap", 0, 0, "bitmap", 0, 0));
```

Note that the widget name (*ed_name*) of the bitmap mask is "bitmap", while that for the object mask that encapsulates it is "object_bitmap". If these defaults are not suitable, or in order to set particular resources for this mask, the `lk_specific` mask may be used.

The system-supplied method `menu` is also redefined to show the three methods: `loadfile`, `display` and `refresh_all`. The `refresh_all` method is particularly useful after calling `loadfile` to create or change a bitmap display or an icon; invoking `refresh_all` updates every instance of the bitmap on screen.

Class `Bitmap`: **schema**

```
class Bitmap public
  type
    tuple(width: integer,
           height: integer,
           bitmap: bits)
  method
    error_msg (label: string),
    public loadfile(filename: string): integer,
    public display,
    public menu: list (string),
    public get_mask: integer
end;
export schema class Bitmap;
```

2.5 The class `Image`

The O₂Look schema has a class called `Image` which you can import into any other schema with the O₂ command:

```
import schema o2kit class Image
```

The `display` method attached to this class uses the specific editor `image`.

Methods of the class `Image`

`loadfile` (*filename*)

This method reads a Unix file containing GIF image data and initializes the value of the receiver object with the image data it has read.

The class Image

- **filename**: a string containing the full Unix path and name of the image GIF file to be read.

display

The generic method `display` is redefined to use the specific editor `image`.

get_width

This method returns the width in pixels of the image contained in the receiver object.

get_height

This method returns the height in pixels of the image contained in the receiver object.

The specific editor `image`

A value of type `image` can be displayed using the usual tuple and list masks. In order to make use of the special features of a `image` value, however, you must create a specific editor mask using the `lk_specific` mask function. For the `image` specific editor, you must call `lk_specific` in the following way:

```
lk_specific (ed_name, rcount, resources, "image", 0, 0)
```

The arguments of the `lk_specific` mask function are as follows:

- **ed_name**: a string representing the widget mask name. This name is used in a resource file specifying the graphic resources for this individual mask.
- **rcount**: an integer count of the number of special resources given in the `resources` argument.
- **resources**: a pointer to an array of graphic resources to be customized in the mask (unless `rcount` is zero). Each resource in the array is of type `Lk_resource`. A list of resources for the image editor is given below.

- **"image"**: the name of the specific editor to create.

image resources

The following table lists the graphic resources which may be customized for any `image` mask; these resources are specified using the `lk_specific` function, either in a resource file in association with the `ed_name` argument, or directly in the `resources` argument of the function.

Name	Type	Default	Access
<code>imageAlignment</code>	<code>char</code>	<code>LkALIGNMENT_CENTR</code>	<code>C</code>
<code>expand</code>	<code>int</code>	<code>1</code>	<code>C</code>
<code>fitInWindow</code>	<code>Boolean</code>	<code>False</code>	<code>C</code>
<code>noinstall</code>	<code>Boolean</code>	<code>False</code>	<code>C</code>
<code>perfect</code>	<code>Boolean</code>	<code>False</code>	<code>C</code>
<code>noglobal</code>	<code>Boolean</code>	<code>False</code>	<code>C</code>
<code>nbcols</code>	<code>int</code>	<code>-1</code>	<code>C</code>

`nimageAlignment`

To specify how the image is aligned. The default centers the image (`LkALIGNMENT_CENTER`). Other possible values are `LkALIGNMENT_BEGINNING` and `LkALIGNMENT_END`.

`expand`

To specify an initial expansion or compression factor for the image as an integer value. Values larger than 1 multiply the image's dimensions by the factor given, i.e. an expansion factor of '3' makes a 320x200 image display as 960x600. Factors less than zero are treated as reciprocals, i.e. an expansion factor of '-4' makes the picture 1/4th its normal size. '0' is not a valid expansion factor.

`fitInWindow`

The class Image

If this resource is True, the image is reduced to fit into the window. For instance, if a 2000x1800 image is to be displayed on a 800x600 display, the image is reduced to 1000x900 and as it is still too big, it is reduced to 500x450.

noinstall

To prevent the image editor from “installing” its own colormap, when the *perfect* resource is True. Instead of installing the colormap, it asks the window manager to do it.

perfect

To make the image editor try to get all the colors it requires. When the *perfect* resource is True, the image editor allocates and installs its own colormap if (and only if) it is unable to allocate all the desired colors.

noglobal

Adjusts the way the editor behaves when it is unable to get all the colors it requested. Normally, it searches the display’s default colormap, and “borrows” any colors it thinks appropriate. These borrowed colors are, however, NOT owned by O₂, and as such, can change without O₂’s permission or knowledge. If this happens, the displayed picture changes in a way you may not desire. If the *noglobal* resource is True, O₂ does not use “global” colors. It only uses colors that it has successfully allocated, thereby making it immune to any color changes.

Please note that “use global colors” is the default because color changes are not generally a problem if you only use xv to display a picture for a short time. Color changes only really become a problem if you use O₂ to display a picture that you are keeping to one side while you do some other work.

nbcols

Sets the maximum number of colors that the image editor uses. Normally, this is set to “as many as it can get” i.e. $2^{(\text{depth of screen})}$. However, you can set this to smaller values for a more interesting effect. Most notably, if you set it to '0', it displays the picture by dithering with 'black' and 'white'.

```
typedef struct {  
    char *sysdir;  
    char *sysname;  
    char *svname;  
    char *swapdir;  
    char **libpath;  
    char **libname;  
}  
O2_sinit;
```

An O₂ installation directory

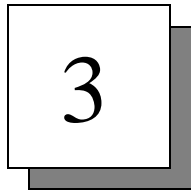
An O₂ named system

Machine where o2server is running

Special swap directory

Paths to search for libraries

Library names to search for



The Hyper Facility

This chapter is divided into the following sections:

- [Introduction](#)
- [The background](#)
- [Data structures](#)
- [Presentation and display](#)
- [Programming example](#)
- [Interactions](#)

3.1 Introduction

A *hyper editor* is any graphic display on which various groups of icons may be superimposed. The icons serve as active buttons for the user of a hyper editor; clicking on an icon triggers an action.

O₂Look provides a specific editor called *hyper* which may be invoked to perform these functions. The *hyper* specific editor is a mask that may be used to display an O₂ value of a particular structure; the value specifies the background display as well as one or more groups of overlayable icons. If permission is given, the user may move icons around, or delete them, or include new ones.

In theory, the background display of a *hyper* mask may be anything: a text display onto which an arbitrary group of hypertext buttons is to be placed, or a bitmap image (a map of a city, for example) on which certain spots are to be sensitized. This latter example is used in what follows.

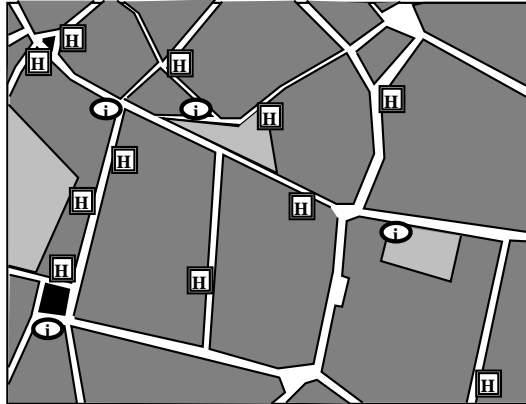


Figure 3.13: A *hyper* display of a city map.

Figure 3.1 shows a city map onto which two different groups of icons have been superimposed. The



icons represent hotels, and the



icons represent information centers. In fact, these are full-fledged O₂ object icons; clicking the right mouse button on a hotel icon, for example, will bring up the menu for the object (of class `Hotel`) corresponding to

Introduction

that spot on the map; selecting `display` from the menu will bring up a full display of the hotel object.

This particular *hyper* display is made up of three parts: a background bitmap (the city map) and two superimposed layers, one each for hotels and information desks. The display may be thought of as three superimposed slides (Figure 3.2 below). The city map is called, logically enough, the *background* of the display, and each of the superimposed layers are called *planes*. Our example has two planes, but there may be any number of them (including zero).

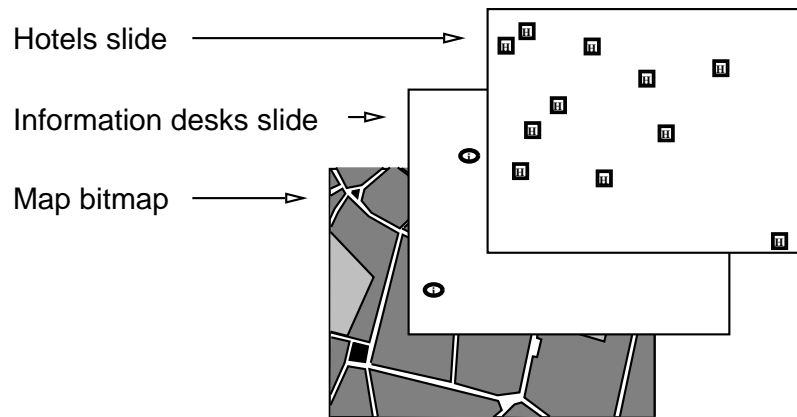


Figure 3.14: Decomposition of a *hyper* display.

3.2 The background

The background display of a `hyper` editor may be anything at all, at least in theory. In practice, backgrounds tend to be either bitmap images (in which case the whole mask is called a *hyper-bitmap*), or else text (called *hypertext*)¹.

The background may be given its own set of coordinates and units of measurement, thus avoiding the necessity of giving absolute pixel locations for the overlaid icons. For example, the map in [Figure 3.13](#) might be a small section of a much larger map of the city in question; the top left corner of the background might really be 5.8 kilometers east and 11.7 kilometers south of the *point of reference* of the larger map. Since it will likely be much more useful to express icon locations in terms of their overall map coordinates (and in kilometers or other map units, rather than in screen pixels), the `hyper` data structure allows the programmer to define these things. The units of measurement and the offset of the background within the larger map may both be specified. The `hyper` editor itself then does all necessary conversions to the pixel locations required by the screen display manager. This is, of course, all voluntary; for those situations where pixel locations are suitable, the pixel may be used as the unit of measurement, and the offsets may be left zero.

[Figure 3.15](#) illustrates an example of coordinate specification.

1. A limitation on the use of the `hyper` editor, at least in its current version, with text is that the superimposed icons do not scroll along with the text. This `hyper` editor is therefore suitable only for fixed, full-page hypertext displays.

3.3 Data structures

The `hyper` editor may be used to display any `O2` value whose type conforms to the following tuple specification.

```
tuple (x:      integer,  
      y:      integer,  
      w:      integer,  
      h:      integer,  
      width:   integer,  
      height:  integer,  
      planes:  list (tuple (name:   string,  
                           visible: boolean,  
                           items:   list (tuple (x:      integer,  
                                                y:      integer,  
                                                object: Object)))))  
      background: <ANY>)
```

The details on each of the attributes is as follows:

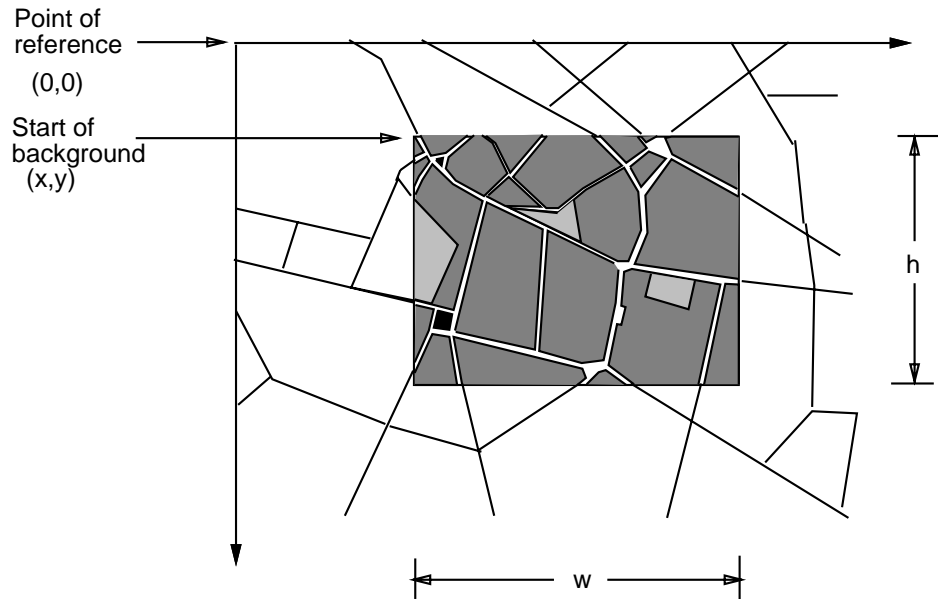


Figure 3.15: Background map as an offset within a larger view.

- `x`: The horizontal offset of the background within a larger frame of reference. This is the distance, in any horizontal unit of measurement, from the `x`-axis of the larger frame to the left edge of the background display.
- `y`: The vertical offset of the background within a larger frame of reference. This is the distance, in any vertical unit of measurement, from the `y`-axis of the larger frame to the top edge of the background display.
- `w`: The width of the background display, expressed in the same units of horizontal measurement as used for `x`.
- `h`: The height of the background display, expressed in the same units of vertical measurement as used for `y`.
- `width`: The width of the background display — the same as `w`, except expressed in pixels. The `hyper` editor uses this information to determine how to convert from the user-defined horizontal coordinates to pixels.
- `height`: The height of the background display — the same as `h`, except expressed in pixels. The `hyper` editor uses this information to determine how to convert from the user-defined vertical coordinates to pixels.
- `planes`: A list of superimposable icon planes. Each element in the list represents one plane; each plane is specified by a tuple containing the following elements:
 - * `name`: The name of the plane. This name appears on the top line (control panel) of the `hyper` display. (Refer to Figure 3.4 for an example).
 - * `visible`: A boolean value indicating whether the icons of this particular plane are displayed or not. In general, the user may modify the displayability of the planes interactively, as described in [Section 3.6](#).
 - * `items`: A list of icons belonging to this plane. Each icon in the list is itself represented by a tuple with three attributes:
 - `x`: The horizontal position of the icon, with respect to the overall frame of reference and in the same units of horizontal measurement used in `x` and `w` above.
 - `y`: The vertical position of the icon, with respect to the overall frame of reference and in the same units of vertical measurement used in `y` and `h` above.
 - `object`: The `O2` object to be represented in the icon. `O2` values may also be specified here, but remember that the rules of subtyping require that all elements of a list must be of compatible types. (Type compatibility is explained in the *O₂C Reference manual*.) By specifying objects of the root class `Object`, this limitation is overcome.

Presentation and display

- `background`: The background of the `hyper` display. This may be of any displayable type.

An example of a `hyper` value appears in [Section 3.5](#).

3.4 Presentation and display

A value of type `hyper` may, of course, be displayed with the usual tuple and list masks. In order to make use of the special features of a `hyper` value, however, a specific editor mask must be created using the `lk_specific` mask function, as explained in the [O2Look User manual](#). For the `hyper` specific editor, `lk_specific` must be called as follows:

```
lk_specific (ed_name, rcount, resources, "hyper", mcount, masks)
```

Such a `hyper` mask may *only* be invoked upon a value whose structure conforms to that described in the *O2Look Manual*. The arguments of the `lk_specific` mask function are as follows:

- `ed_name`: a string representing the widget name of the mask. This name may be used in a resource file to specify graphic resources for this individual mask.
- `rcount`: an integer count of the number of special resources given in the `resources` argument.
- `resources`: a pointer to an array of graphic resources to be customized in the mask (unless `rcount` is zero). Each resource in the array is of type `Lk_resource`. A list of resources for the `hyper` editor is given below.
- `"hyper"`: the name of the specific editor to create.
- `mcount`: the number of sub-masks used by this specific editor. This is the number of planes plus 1 (for the background).
- `masks`: a pointer to an array of sub-masks (i.e. values of type `Lk_mask`) used in this specific editor. The number of masks in this array is `mcount`. The first mask in the array will be used to display the background; the second (if any) is for the object icons of the first plane; the third is for the second plane; and so forth.

Note

For details of how to use graphic resources, please refer to the [O2Look User manual](#).

The following table lists the graphic resources which may be customized for a *hyper* mask using the `lk_specific` function, either in a resource file in association with the `ed_name` argument, or directly in the *resources* argument of the function.

Name	Type	Default	Access
foregroundColor	Color	Dynamic	CS
backgroundColor	Color	Dynamic	CS
copyProcess	Boolean	True	CS
cutProcess	Boolean	True	CS
insertProcess	Boolean	True	CS
fontList	Font	"fixed"	CS
orientation	unsigned char	VERTICAL	CS
visibleControl	Boolean	True	CS
maxWidthVisible	unsigned short	300	C
maxHeightVisible	unsigned short	300	C

- **foregroundColor**

Specifies the foreground color for the *hyper* mask.

- **backgroundColor**

Specifies the background color for the *hyper* mask.

- **replaceProcess**

Specifies whether the contents of an item icon can be replaced with a paste operation.

- **cutProcess**

Presentation and display

Specifies whether a cut (removing an item using *Control* plus right mouse button) can be performed. If *True* (default), you can remove icons from the *hyper* display, and the corresponding items are deleted from their respective planes.

- **copyProcess**

Specifies whether a copy (copying an item using *Control* plus middle mouse button) can be performed.

- **insertProcess**

Specifies whether an item can be inserted onto the *hyper* display with a paste operation. If *True* (default), you can paste icons onto the *hyper* display, and new items corresponding to those icons are appended to one plane. Refer to [Section 3.6](#). Note that items may be moved from one location to another by combining the cut process with the insertion process.

- **fontList**

Specifies the font used for the button labels.

- **orientation**

Specifies whether buttons are laid above the image (*VERTICAL* default) or beside it (*HORIZONTAL*).

- **visibleControl**

Specifies whether the visibility of the planes can be modified interactively. Refer to [Section 3.6](#).

- **maxWidthVisible**

Specifies the maximum width of hyper for the application.

- **maxHeightVisible**

Specifies the maximum height of hyper for the application.

3.5 Programming example

As an example, consider the following class definitions:

```
class City
public type tuple (name: string,
                  map: bitmap,
                  map_start: tuple (horiz_coord: integer,
                                    vert_coord: integer),
                  map_end: tuple (horiz_coord: integer,
                                   vert_coord: integer),
                  hotels: list (Hotel),
                  info_centers: list (Info))

method ...
end;
class Hotel
public type tuple (name: string,
                  location: tuple (horiz_coord: integer,
                                   vert_coord: integer,
                                   street_address: string,
                                   city: City),
                  prices: list (tuple (facilities: string,
                                       low_season: Money,
                                       high_season: Money)))

method ...
end;
class Info
public type tuple (location: tuple (horiz_coord: integer,
                                   vert_coord: integer,
                                   attraction: tuple (name: string,
                                                       view: bitmap)))

method ...
end
```

You know from O₂Look that the type structure of a bitmap is the following, so let us define a named type for it.

```
type bitmap: tuple (width: integer,
                   height: integer,
                   bitmap: bits)
```

Programming example

This tells us the width and height of the background bitmap *in pixels*; the attributes `map_start` and `map_end` give us the same information *in human-friendly units*. As long as we are defining named types, the following ones will help us to avoid ungainly nested tuples:

```
type oneitem: tuple (x:      integer,
                    y:      integer,
                    object: Object);
type oneplane: tuple (name:   string,
                    visible: boolean,
                    items:   list (oneitem))
```

Given, say, a named object of class `City` called `Roma`, we may construct and display a value of type `hyper` as follows:

```
program body foo in application bar {
/*
 *   Construct and display a hyper value for the city of Rome
 *
 *   First, define hyper value
 */
  o2 tuple (x:          integer,
            y:          integer,
            w:          integer,
            h:          integer,
            width:      integer,
            height:     integer,
            planes:     list (oneplane),
            background: bitmap)
    hyperthing;
/* Mask and presentation variables, iterators */
  Lk_mask mhyper[3];
  Lk_presentation pl;
  o2 Info ic;
  o2 Hotel h;
/* Set up hyper value from object Roma */
  hyperthing.x = Roma->map_start.horiz_coord;
  hyperthing.y = Roma->map_start.vert_coord;
  hyperthing.w = Roma->map_end.horiz_coord - Roma->map_start.horiz_coord;
  hyperthing.h = Roma->map_end.vert_coord - Roma->map_start.vert_coord;
  hyperthing.width = Roma->map.width;
  hyperthing.height = Roma->map.height;
/* The two planes */
  hyperthing.planes = list (tuple (name: "Info",
                                   visible: true,
                                   items: (o2 list(oneitem)) list()),
```

Programming example

```

                                tuple (name: "Hotels",
                                        visible: true,
                                        items: (o2 list(oneitem)) list()));
/*    The background */
hyperthing.background = Roma->map;
/*    Fill in items for Info plane */
for (ic in Roma->info_centers)
    hyperthing.planes[0].items += list (tuple (x:
                                                y: ic->location.vert_coord,
                                                object: ic));
/*    Fill in items for Hotel plane */
for (h in Roma->hotels)
    hyperthing.planes[1].items += list (tuple (x: h->location.horiz_coord,
                                                y: h->location.vert_coord,
                                                object: h));
/*    Set up mask for background (bitmap) */
mhyper[0] = lk_specific ("hback", 0, 0, "bitmap", 0, 0);
/*    Set up mask for Info plane (object icon) */
mhyper[1] = lk_object ("hinfo", 0, 0, 0);
/*    Set up mask for Hotel plane (object icon) */
mhyper[2] = lk_object ("hhotel", 0, 0, 0);
/*    Define and map the hyper presentation */
p1 = lk_present (hyperthing,
                 lk_specific("hyperroma", 0, 0, "hyper", 3, mhyper));
lk_map (p1, LK_MANUAL, 0, LK_SCREEN, 0, 0);
while (!(lk_wait (p1) == LK_ERASE));
lk_delete_presentation (p1);
}
```

This program would produce a display similar to the one shown in [Figure 3.16](#)

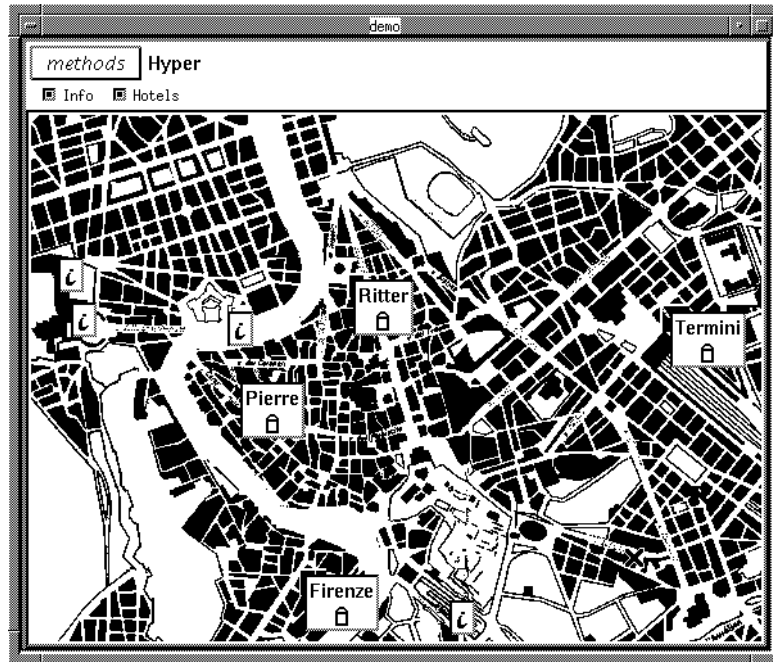


Figure 3.16: A *hyper* display of Rome.

3.6 Interactions

Given the appropriate mask, a `hyper` value is displayed in the following way:

1. The `background` member is drawn, and its size becomes the size of the `hyper` display.
2. For each plane of the `hyper` value whose `visible` boolean attribute is set to true, each object item of the plane is displayed according to the submask specified for that plane, at its appropriate location.
3. If the `visibleControl` resource is set to true (the default), a panel is displayed above the background. This panel contains one toggle button per plane, labeled with the name of the plane (as specified in its `name` attribute). When the user clicks on the toggle button of a displayed plane, its icons disappear and its `visible` attribute is changed to false; when the user clicks on the toggle button of an invisible plane, its icons are displayed and its `visible` attribute is changed to true.

By clicking on both of the visibility buttons in the control panel of [Figure 3.16](#), both planes disappear and only the background remains; see [Figure 3.5](#) below for the result.

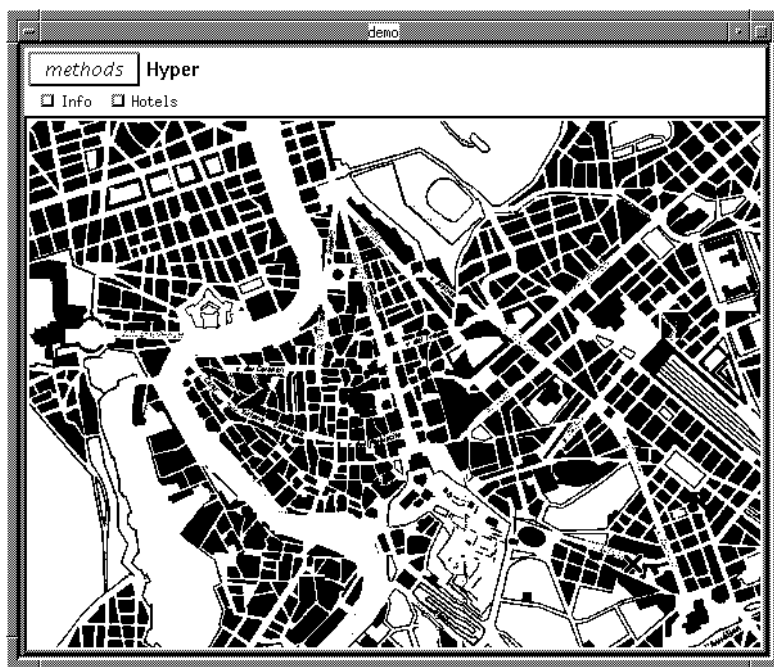


Figure 3.17: Both planes invisible.

In addition to toggling the visibility of the planes, the user is able to execute the following interactions:

- Interact normally with the background display or any of the items. In particular, since all items are normal O₂Look icons, the user can call up the icon menu and trigger methods (display, for instance) on the objects represented. In Figure 3.6 below, we have invoked the display method on the information center icon near the Vatican.

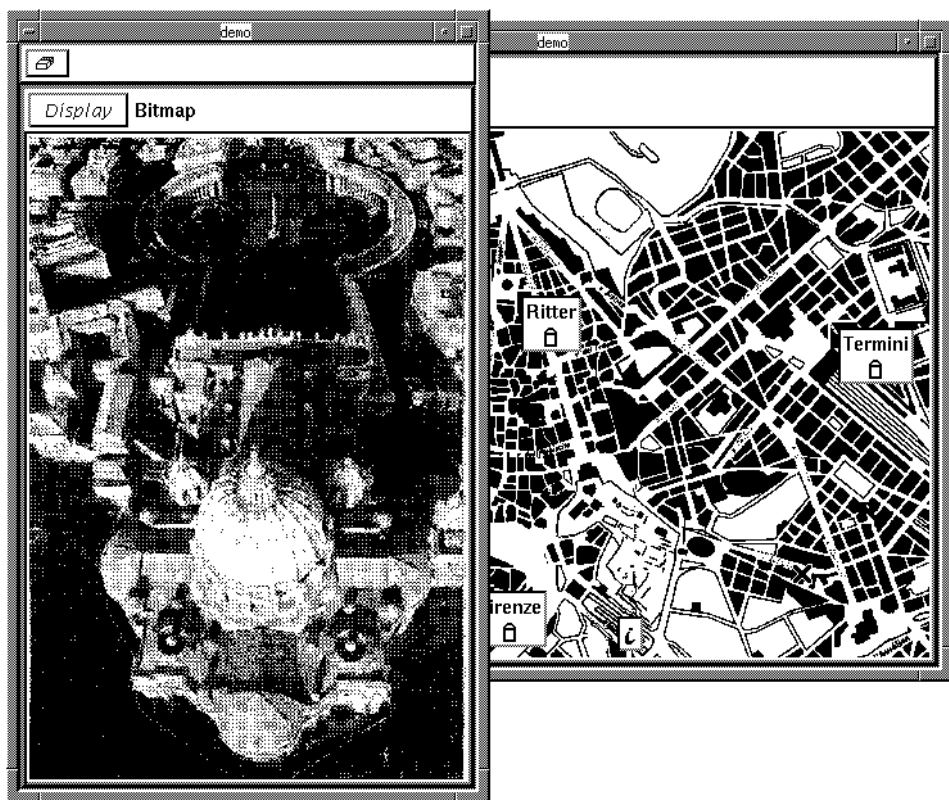


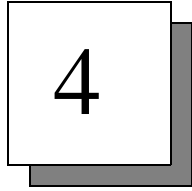
Figure 3.18: Invoking a method on an item of the Info plane.

- Remove an item from the display, move an item from one location to another, replace an item by another, or insert a new item. These operations are performed through the usual copy, cut and paste processes of O₂Look . All changes in the appearance of the hyper display are automatically reflected in the hyper value used for the display. For example, if an icon is removed from the display, the item corresponding to that icon is deleted from the list of items in the plane; if an icon is moved, its *x* and *y* coordinates are changed.

Interactions

These processes may be inhibited for a particular `hyper` mask through the specification of graphic resources; refer to [Section 3.4](#) above.

Restriction: In the case of pasting an item onto the `hyper` display, the item is *always* inserted into the last plane whose visibility has been toggled to `true`. In other words, only one plane at a time is open for insertions, and the user must identify that plane by toggling its visibility to `true`.



The Widget Editor

This chapter is divided into the following sections :

- [Introduction](#)
- [The class WidgetDialoger](#)
- [Methods of the class WidgetDialoger](#)
- [An example of the user interface](#)

4.1 Introduction

The **Widget Editor** is a specific O₂Look editor. It enables you to insert any Motif window into an O₂Look presentation.

As a Motif programmer, you can use this editor to implement each of the functions of the O₂Look protocol as you wish. Functions such as `lk_present`, `lk_consult`, `lk_refresh_presentation` and `lk_delete_presentation` - refer to the relevant O₂Look documentation.

You can use the Widget editor to insert very elaborate interfaces that have been generated by interface generators, and to display video images.

You do this by associating a function of your own to an event that can occur in O₂Look . This allows you to totally control your Motif window no matter what events are sent to it from O₂Look e.g. `lk_present`, `lk_consult`, etc.

This association is carried out using a dialoguer in the form of an O₂ class `WidgetDialoguer`. With this dialoguer, you attach to each event of the Motif window one of your own O₂C function. The dialoguer therefore calls your function each time an O₂Look editor protocol function should have been called.

Note

Any O₂ class can be used with the Widget editor.

4.2 The class `WidgetDialoguer`

The `WidgetDialoguer` class has the following syntax:

Methods of the class WidgetDialoger

```
class WidgetDialoger
private type tuple(editor: integer,
                  active: boolean,
                  triggers: list(tuple(event: integer,
                                     target: integer,
                                     method: string)))

method public initialize : string,
        public set_trigger_event (event: integer,
                                target: Object,
                                method_name: string)

/* recovery of inherited resources */
get_backgroundColor: integer,
get_foregroundColor: integer,
get_editable: boolean,
end;
```

The attributes of the Widget editor correspond to an identity card of the O₂Look presentation that is manipulated by the WidgetDialoger methods described in [Section 4.3](#) below.

4.3 Methods of the class WidgetDialoger

The editor attribute identifies the O₂Look presentation to be manipulated by the WidgetDialoger methods. The initialize method sets up this link. The active attribute, as the name implies, distinguishes between mapped and unmapped presentations. The triggers attribute is a list of events and the methods that they invoke. This list is specified through the set_trigger_event method. The WidgetDialoger class has the following methods:

- **initialize: string**

Initializes the widget editor and returns a character string that must be placed as the value of the dialoger resource.

- **set_trigger_event(event: integer, target: Object, method_name: string)**

This method establishes a link between the presentation event and your method or function that is managing it. Once this link has been set up, the method or function is called transparently as soon as this event is generated.

The following table is an exhaustive list of these events.

Event Number	Type of Function	O ₂ signature
0	lk_present	(father : integer)
1	lk_consult	(top : integer)
2	lk_refresh_presentation	(top : integer)
3	lk_delete_presentation	()

0

The method attached to this event is called when a call to `lk_present` is reached by the program. You must create the widget hierarchy of your editor in the method. The father widget to which you must attach the widget hierarchy is given in the method argument.

1

The method attached to this event is called when a call to `lk_consult` is reached by the program. It enables you to update the database according to the current state of your widgets. The root widget of the hierarchy you created in response to the creation event is given in the method argument.

2

The method attached to this event is called when a call to `lk_refresh_presentation` is reached by the program. It allows you to update your widget hierarchy according to the current state of the database. The root widget of the hierarchy you created in response to the creation event is given in the method argument.

3

The method attached to this event is called when a call to `lk_delete_presentation` is reached by the program. It allows you to free all the resources you have created at creation time.

Warning !

You must not destroy the widgets you have created as they will be destroyed by O₂Look . The following methods are associated to editor resources. Each one enables you to retrieve its associated value.

An example of the user interface

- `get_backgroundColor: integer`

This method returns the resource value specifying the background color of the window.

- `get_foregroundColor: integer`

This method returns the resource value specifying the foreground color of the window when it is active.

- `get_editable: integer`

This method returns the resource value specifying whether or not the window can be edited or not.

The following table lists the resources that you can use in the mask that is associated to the editor. You do this either in the resource file by associating the editor name `widget`, or directly in the `resource` argument of the `lk_specific` function.

Name	Type	Default	Access
<code>backgroundColor</code>	Color	Dynamic	CS
<code>foregroundColor</code>	Color	Dynamic	CS
<code>editable</code>	Boolean	True	CS

4.4 An example of the user interface

This is an example uses an O₂ application and a class `QuestionSubsidiary` of the user schema. You want to display a Motif window in the O₂Look presentation that is associated to the object `question` of the class `QuestionSubsidiary`.

```
class QuestionSubsidiary /* user class and methods */
private type ...
method public createQuestionSubsidiary, /* creation of Motif widget */
      public consultQuestionSubsidiary /* recovery of the response */
end;
```

The method `createQuestionSubsidiary` creates a Motif window, with a field `text` and a title.

```
method body createQuestionSubsidiary(father: integer): integer in class
QuestionSubsidiary
{
#include <stdio.h>
#include <Xm/Xm.h>
Widget fenetreWE;
Arg args[5];
int n=0;
self->thequestion = "who are you ?";
/* create form */
fenetreWE = XmCreateForm( (Widget)father, "Forme", args, n);
n= 0;
rowcol = XmCreateRowColumn((Widget)top, "rowcol", args,n);
XtManageChild(rowcol);
n = 0;
text_label= XmStringCreateLtoR(name, charset);
XtSetArg(args[n],XmNlabelString, text_label); n++;
label= XmCreateLabel (rowcol, "label", args,n);
XtManageChild(label);
n = 0;
XtSetArg (args[n], XmNcolumns, columns); n++;
val= (char *) malloc (256); strcpy(val, thequestion);
XtSetArg(args[n], XmNvalue, val); n++;
prompt= XmCreateText(rowcol,"text",args,n);
widgetText = (o2 integer)prompt;
XtManageChild (fenetreWE);
printf("createQS:: OK\n");
return (o2 integer)fenetreWE;
};
```

The method `consultQuestionSubsidiary` consults the field `Text` that was created before.

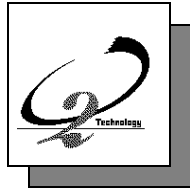
```
method body consultQuestionSubsidiary in class QuestionSubsidiary
{
#include <Xm/Xm.h>
#include <Xm/Text.h>
char * answer;
answer = XmTextGetString((Widget)widgetText);
self->theresponse = (o2 string)answer;
};
```

An example of the user interface

These two public methods can be called directly by the program. The body of the application has the following format:

```
run body {
    Presentation p;
    Mask mask = lk_generic();
    Mask w_mask, obj_mask;
    Lk_resource w_ress[1];
    o2 QuestionSubsidiary question = new QuestionSubsidiary;
    o2 integer x;
    /*--- dialoger of the widget ---*/
    o2 WidgetDialoger w_dialog = new WidgetDialoger;
    o2 string str_dialog;
    str_dialog = w_dialog -> initialize;
    w_ress[0].name = ``dialoger``;
    w_ress[0].value = str_dialog;
    w_dialog -> set_trigger_event( create, question,
``createQuestionSubsidiary`` );
    w_dialog -> set_trigger_event( consult, question,
``consultQuestionSubsidiary`` );
    /*--- mask of the presentation ---*/
    w_mask = lk_specific( ``geneWidget``, 1, w_ress, ``widget``, 0, 0 );
    obj_mask = lk_object( ``geneObject``, 0, 0, w_mask );
    /*--- The presentation ---*/
    p = lk_present( question, obj_mask, 0,0,0 );
    lk_map( p, MANUAL, 0, 0, 0, 0 );
    x = lk_wait(p);
    if( x == LK_SAVE )
        lk_consult(p, question);
    lk_delete_presentation(p);
}
```

`createQuestionSubsidiary` and `consultQuestionSubsidiary` are public methods of the class `QuestionSubsidiary`. These methods are called if the events `create` and `consult` are respectively generated. Note that these events are respectively called by `lk_present` and `lk_consult`, amongst others. `geneWidget` and `geneObject` are the two names that are respectively attached to the specific editor called `ed_widget` that is associated to the mask `w_mask`, and to the object associated to this same mask `w_mask`. They enable their resources to be initialized by file.



INDEX



A

add_days [18](#)
add_months [18](#)
add_years [18](#)
Architecture
 O₂ [10](#)

B

backgroundColor [48, 49, 50, 51, 52, 53, 54, 76](#)
bitmap
 specific editor [62, 63](#)
Bitmap class. See Class
 Bitmap
bitmap image [63](#)
Box class. See Class
 Box
Button class. See Class
 Button
Button_box class. See Class
 Button_box

C

C [11](#)
C++ [11](#)
Class
 Bitmap [62](#)
 methods [63](#)
 schema [64](#)
 Box [26](#)

 Methods [27](#)
Button [34](#)
 Resources [48](#)
Button_box [36](#)
 Resources [49](#)
Component [33, 41](#)
Date [16](#)
 mask [22](#)
 methods [16](#)
Dialog_box [41](#)
 methods [41](#)
Editable_selection [37](#)
Image [64](#)
 Methods [64](#)
Label [40](#)
Multiple_selection [38](#)
Picture [40](#)
Prompt [39](#)
Radio_box [35](#)
 Resources [49](#)
Single_selection [37](#)
Text [58](#)
 Methods [59](#)
WidgetDialoger [88](#)
columns [52](#)
Component class. See Class
 Component
copyProcess [77](#)
create_presentation [19, 33](#)
cutProcess [76](#)

D

Date class. See Class
 Date
dialog [29](#)
Dialog_box class. See Class
 Dialog_box
Dialogue boxes [26, 58](#)
Dialoguer object [43](#)
diff [18](#)
display [19, 60](#)

INDEX

E

`edit` 19, 60
`edit_selection` 31
Editable_selection class. See Class
 Editable_selection
Editor 91
 hyper. See Hyper editor
 O₂Look 88
 resources 90
 specific. See also Specific editor
 widget. See also Widget editor
Elements 41

F

`fontList` 48, 50, 51, 52, 53, 77
`foregroundColor` 48, 49, 50, 51, 52, 53, 54, 76
Function
 `lk_consult` 34, 88, 90, 93
 `lk_delete_presentation` 88, 90
 `lk_method` 47
 `lk_present` 44, 47, 88, 90, 93
 `lk_prologue` 27
 `lk_refresh_presentation` 88, 90
 `lk_specific` 20, 22, 60, 63, 65, 66, 75, 76,
 91

G

`get_backgroundColor` 91
`get_day` 17
`get_foregroundColor` 91

`get_mask` 20, 34, 60, 63
`get_month` 17
`get_year` 17
Graphic resources 48
 Button 48
 Button_box 49
 Editable selection 51
 Label 53
 Multiple selection 50
 Picture 53
 Prompt 52
 Radio_box 49
 Single_selection 49

H

Hyper editor 70, 72, 73, 74, 75
 hyper-bitmap 72
 hypertext 72
Hyper editor. See also hyper in Specific
 editor

I

image
 specific editor 64, 65
Image class. See Class
 Image
`import schema` 16
`init` 33, 41
`initialize` 89
`insertProcess` 77



INDEX

J

Java [11](#)

L

Label. See also Class
Label

`labelPixmap` [48, 54](#)

`labelString` [49, 53](#)

`labelType` [48](#)

`libname` [68](#)

`libpath` [68](#)

`lk_*`. For all `lk_*` functions see
Functions

`Lk_*`. For all `Lk_*` value type see Value
Type.

`loadfile` [63](#)

M

Mask [19, 20, 23, 41, 47, 59, 60, 63, 65, 75, 83](#)

 bitmap mask [63](#)

 date mask [19, 20, 20](#)

 hyper mask [70, 72, 75, 76, 85](#)

 image mask [66](#)

 list mask [65, 75](#)

 object mask [20, 60, 63](#)

 specific editor [23, 34, 44, 65, 75, 91](#)

 submask [23, 47, 75, 83](#)

 text mask [60](#)

 tuple mask [22, 22, 75](#)

`w_mask` [93, 93](#)

`maxHeightVisible` [77](#)

`maxWidthVisible` [77](#)

`message` [27](#)

Methods

 Bitmap [63](#)

 Box [27](#)

 Date [16](#)

 Image [64](#)

 Text [59](#)

`mult_selection` [32](#)

Multiple_selection class. See Class
Multiple_selection

N

`new Button` [34](#)

`new Date` [17](#)

O

O₂
 Architecture [10](#)

O₂C [11](#)

O₂Corba [11](#)

O₂DBAccess [11](#)

O₂Engine [10](#)

O₂Graph [11](#)

O₂Kit [11](#)

O₂Kit Schema [16](#)

O₂Look [11](#)

 editor [88](#)

 mask [22](#)

 resources [48](#)

O₂ODBC [11](#)

O₂Store [10](#)

O₂Tools [11](#)



INDEX

O₂Web [11](#)

OQL [11](#)

orientation [49, 77](#)

P

Picture. See also Class
Picture

Prompt. See also Class
Prompt

Q

question [28](#)

R

Radio_box class. See Class
Radio_box

read_file [59](#)

refresh_all [63](#)

replaceProcess [76](#)

Resource file [46, 48, 65, 66, 75, 75, 76, 91](#)

Resources. See also Graphic Resources

S

selection [30](#)

Separators [20](#)

set_day [17](#)

set_month [17, 18](#)

set_to_current_date [18](#)

set_trigger_event [89](#)

Single selection
Resources [50](#)

Single_selection class. See Class
Single_selection

Specific editor [16, 20, 20, 59, 60, 66, 75, 93](#)
date [19](#)
hyper. See also hyper editor
image [65](#)

svname [68](#)

swapdir [68](#)

sysdir [68](#)

sysname [68](#)

System
Architecture [10](#)

T

Text class. See Class
Text

titleFontList [49, 50, 51, 52](#)

to_date [21](#)

to_string [21](#)

V

Value type
Lk_mask [34, 75, 75](#)
Lk_presentation [33, 42](#)
Lk_resource [23, 65, 75, 93](#)

visibleControl [77](#)

visibleItemCount [50, 51](#)



INDEX

`visibleItemCountStatic` [50, 51](#)

W

Widget editor [87, 88, 89](#)

WidgetDialoger class. See Class
WidgetDialoger

`write_file` [59](#)